KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Formalisation and Soundness of Static Verification Algorithms for Imperative Programs

Frédéric VOGELS

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

December 2012

Formalisation and Soundness of Static Verification Algorithms for Imperative Programs

Frédéric VOGELS

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

December 2012

D/2012/7515/139
ISBN 978-94-6018-605-9

Abstract

Not only does our software grow larger and more complex, we also become more dependent on it, thus making it all the more necessary to develop tools that assist us in writing correct programs. As a consequence, much research has been done in the field of static verification, i.e. the development of algorithms that analyse source code and determine whether it contains certain kinds of errors. This can range from checking that no null dereferences can occur at runtime to full functional correctness.

Verification algorithms, however, are just as much subject to mistakes. Therefore, it is important to put these algorithms under scrutiny: our trust in software can only be as strong as the confidence we can have in our verification tools. In a first step, we closely examine some existing approaches to verification. More specifically, we fully formalise the verification algorithms and prove their soundness.

If we are not willing to trust our software nor our verification algorithms, one can wonder why we should trust our formalisations and proofs. For this reason, we also provide full Coq implementations of all verification algorithms we consider, and, for most of them, machine check the soundness proofs.

This thesis is divided into two parts. In the first part, we discuss verification condition generation. This approach consists of deriving a logical formula (called verification condition) from a program's source code, whose validity implies the correctness of the program with respect to given specifications. Three such algorithms are investigated, namely the strongest postcondition, the weakest liberal precondition and the weakest precondition.

Extra attention is given to the weakest precondition algorithm. In its classic form, it produces a verification condition that grows exponentially with respect to the size of the program. An alternative formulation is available which generates a verification condition that grows only polynomially, but requires the program to be passive, i.e. to not contain any assignments. Fortunately, it is possible to transform any program into an equivalent passive form. We implement this transformation in Coq as well as the more efficient variant of the weakest precondition algorithm, and we provide fully machine checked proofs that this approach is sound.

In the second part of the thesis, we turn our attention to symbolic execution. This approach consists of abstractly interpreting the program in such a way that all possible execution paths are considered simultaneously. Verification succeeds if no errors are encountered during this execution.

Based on this technique we develop Featherweight VeriFast, representing the core part of VeriFast, an existing verifier developed at the KU Leuven. Featherweight VeriFast is formally defined as a denotational semantics, but it has been implemented in Coq and is extractable, making it usable as a standalone verifier.

Featherweight VeriFast's formalisation is built on top of three abstraction layers. The first layer, the result algebra, allows us to reason about angelic and demonic choices, both needed to express the result of a symbolic execution. The second layer defines operators, composable monadic functions which allow us to elegantly deal with the two kinds of nondeterminism and state in a purely functional setting. The third layer provides basic operators, which together form a domain specific language, abstracting away the details of the result algebra. The symbolic execution, central to Featherweight VeriFast's operation, is then defined in terms of these basic operators. Finally, we also provide a (partially machine checked) soundness proof for Featherweight VeriFast.

A last chapter discusses verification automation techniques. Some verifiers, for example VeriFast, require the code to be annotated (with for example routine contracts or loop invariants) to be able to determine a program's correctness. Generating these annotations automatically can dramatically decrease the effort required to verify programs. We discuss and compare three different automation techniques. Lastly, we propose a framework in which automation techniques can be added without compromising the soundness of the verification.

Abstract (Nederlands)

Niet alleen wordt onze software alsmaar omvangrijker en complexer, we hangen er eveneens steeds meer van af, wat het des te noodzakelijker maakt hulpmiddelen te ontwikkelen om ons bij te staan bij het schrijven van correcte programma's. Bijgevolg werd veel onderzoek gevoerd in het domein van statische verificatie, d.i. het uitwerken van algoritmes dewelke broncode analyseren en uitmaken of het bepaalde types fouten bevat. Dit kan variëren van het nakijken dat er geen *null dereferences* gebeuren tijdens de uitvoering van het programma tot het verifiëren van volledige functionele correctheid.

Verificatiealgoritmes zijn echter eveneens onderhevig aan fouten. Daarom is het belangrijk om deze algoritmes nauwkeurig te onderzoeken: het vertrouwen dat we hebben in onze software kan niet sterker zijn dan dat in onze verificatiehulpmiddelen. In een eerste stap bekijken we verschillende benaderingen tot verificatie; meer in het bijzonder formaliseren we de algoritmes en bewijzen we hun correctheid.

Indien we niet bereid zijn onze software noch onze verificatiealgoritmes te vertrouwen, kunnen we ons afvragen waarom we onze formalisaties en correctheidsbewijzen wel zouden vertrouwen. Om deze reden implementeren we in een tweede stap alle door ons beschouwde verificatiealgoritmes in Coq en bieden we in de meeste gevallen ook machinaal nagekeken correctheidsbewijzen aan.

Deze thesis bestaat uit twee delen. In het eerste deel behandelen we het genereren van verificatiecondities. Dit houdt in dat we op basis van de broncode van een programma een logische formule genereren waarvan de validiteit impliceert dat het programma correct werkt ten opzichte van gegeven specificaties. We onderzoeken drie zulke algoritmes, namelijk de sterkste postconditie, de zwakste vrije preconditie en de zwakste preconditie.

Extra aandacht wordt besteed aan het zwakste preconditie algoritme. In zijn klassieke vorm genereert het een verificatieconditie die exponentieel groeit in de grootte van het programma. Er bestaat echter wel een alternatieve formulering die polynomiale verificatiecondities produceert, maar deze kan enkel gebruikt worden op passieve programma's, met andere woorden programma's die geen toekenningen bevatten. Het is gelukkig mogelijk om willekeurige programma's om te zetten naar een equivalente passieve vorm. We implementeren deze programmatransformatie samen met de efficiëntere vorm van het zwakste preconditie algoritme in Coq, waarna we machinaal bewijzen dat deze aanpak correct is.

In het tweede deel van de thesis behandelen we symbolische uitvoering. Dit bestaat erin om een programma op abstracte wijze te interpreten zodat alle uitvoeringspaden tegelijkertijd beschouwd worden. Verificatie slaagt indien men geen fouten tegenkomt tijdens deze uitvoering.

Op basis van deze techniek ontwerpen we Featherweight VeriFast, dewelke de kern voorstelt van VeriFast, een bestaand verificatieprogramma dat ontwikkeld wordt aan de KU Leuven. Featherweight Verifast wordt formeel gedefinieerd als een denotationele semantiek, maar vermits het geïmplementeerd werd in Coq en extraheerbaar is, is het eveneens bruikbaar als verificatieprogramma.

De formalisatie van Featherweight VeriFast steunt op drie abstractielagen. In een eerste laag definiëren we de resultatenalgebra. Deze laat ons toe te redeneren over demonisch en angeliek nondeterminisme, dewelke nodig zijn om de resultaten van symbolische uitvoering uit te drukken. De tweede laag definieert operatoren: dit zijn samenstelbare monadische functies die ons toelaten om op elegante wijze om te gaan met de twee types nondeterminisme en bieden ons toestand aan binnen een zuiver functionele context. De derde laag definieert de basisoperatoren. Samen vormen deze een domeinspecifieke taal die de details van de resultatenalgebra wegabstraheert. De symbolische uitvoering, de centrale component van Featherweight VeriFast, wordt dan uitgedrukt in termen van deze basisoperatoren. We geven ook een (gedeeltelijk machinaal nagekeken) correctheidsbewijs van Featherweight VeriFast.

Het laatste hoofdstuk bespreekt verificatie-automatiseringstechnieken. Sommige verificatieprogramma's, waaronder VeriFast, vereisen dat de code geannoteerd wordt met bijvoorbeeld routinecontracten en lusinvarianten. Het automatisch genereren van deze annotaties kan de hoeveelheid werk dat nodig is om een programma te verifiëren drastisch doen afnemen. Om af te sluiten stellen we een algemeen kader voor dat het mogelijk maakt om nieuwe automatiseringstechnieken toe te voegen zonder daarbij de betrouwbaarheid van de verificatie in het gedrang te brengen.

Acknowledgements

I wish to thank my supervisor Prof. Dr. ir. Frank Piessens for his commitment to my cause. His dedication to his PhD students and his sense of responsibility never ceased to impress me and I hope I never abused his qualities as a supervisor. I also thank Prof. Dr. ir. Wouter Joosen for taking a chance on me and accepting me as a PhD student within DistriNet.

I am also grateful to Prof. Dr. ir. Eric Steegmans, whom I have always respected and will always remember as an excellent teacher, as I am sure will his many students. His immense devotion to his students has been a great inspiration to me.

Prof. Dr. Bart Jacobs certainly also deserves a special thanks for all his assistance during my years at DistriNet. Our weekly gatherings and our many talks have turned him into more than just a supervisor. I hope this tradition survives my departure. I also wish him great success with his career at the KU Leuven.

I would also like to thank the other jury members for their valuable time and their many suggestions to improve this thesis: Prof. Dr. Dave Clarke, Prof. Dr. Marieke Huisman and Prof. Dr. ir. Willy Sansen.

Many other people deserve to be thanked for the role they played in my life. I can only (futilely) hope the following list is exhaustive; my apologies for anyone I missed. I inductively thank all my ancestors; without them there would be no PhD. A special thanks goes to my mother; she's everything a son could wish for, and more. Though a single mother, it felt like I had three. *T'es la meilleuse.* I hope I made her proud.

I thank Michel Boghe, my lawyer, my psychologist, my friend. Life would not be the same without you. I thank Bastiaan Boghe for being the cute and smart little guy he is. Do not forget to thank your godfather in your own PhD.

I thank Pim Volders, my fellow traveller on the road to graduation. It was quite a challenge to keep ahead of you. Thank you for choosing to be my teammate during many projects. You could always be counted on.

I thank my office buddies, who have had to put up with my (occasional) presence for years. Yves Younan, thank you for letting me buy your car. Pieter Philippaerts, my partner in crime, thank you for all the great times we had. I count on you to take good care of Freddy Bear. Raoul Strackx, thank you for having endured my extravagances. The extent of my gratitude is unknown to you, but I am working on repaying you. Just eight to go. Mathy Vanhoef, sadly our time together at the university will be short. I wish you good luck with your research. As (non-negligible) contribution to your future career, I donate my part of the whiteboard to you.

I thank Ansar-Ul-Haque Yasar and Nelson Matthys for the great times we had in Switzerland, and Marco Patrignani for the equally great moments in Iceland. I wish to thank the (self-proclaimed) awesome and unforgettable Milica Milutinović for bettering my life: seldom have I met such strict fitness and diet coach.

I thank Marleen Giedts: thank you for your good advice regarding my finances and for your tireless attempts at trying to educate me in the matter. Thank you for giving me a chance, and although the bumpy road has not led us to the destination I had hoped for, we still ended up somewhere nice. Know that it is cherished, and I wish it may last.

I thank Yolande Berbers, I thank you for choosing to have me on your didactic team for four years, for your trust and for the freedom you gave us. I also thank all past and present members of Team Olympus: Ruben Vermeersch, Dominique Devriese, Willem De Groef, Job Noorman and Jasper Bogaerts.

I thank my colleagues for making my time at DistriNet so memorable. I apologise to the people I forget to mention: Pieter Agten, Mario Henrique Cruz Torres, Philippe De Ryck, Lieven Desmet, Francesco Gadaleta, Kristof Geebelen, Bert Lagaisse, Jan Tobias Mühlberg, Nick Nikiforakis, Willem Penninckx, Davy Preuveneers, Jose Paiva Proenca, Rula Sayaf, Ilya Sergey, Jan Smans, Klaas Thoelen, Marko van Dooren, Gijs Vanspauwen, Dries Vanoverberghe and Koen Yskout.

I thank Nancy Mazur, which even more than ten years later has remained unforgotten. I thank you for having opened the eyes of a die hard C++ fan and introduced him to the wonders of functional programming. I have yet to repeat your accomplishment.

I thank all members of the administrative force for their excellent work which is all too often taken for granted: Denise Brams, Inge Vandenborne, Marleen Somers, Karen Verresen, Esther Renson, Liesbet Degent, Karin Michiels and

Margot Peeters.

I thank Katrien Janssens and Ghita Saevels, members of the project office. Ghita, I am looking forward to the excellent waffles you so generously offered to make for my reception.

I thank my students, of which there are too many to mention, for having made my educational duties which were part of my doctoral training more gratifying. I hope most of them will graduate having good memories of me.

This thesis presents the results of research that has been partially funded by the Research Fund KU Leuven and by the EU FP7 project NESSoS, with the financial support from the Prevention of and Fight against Crime Programme of the European Union.

Frédéric Vogels
December 2012

Contents

Contents ix

List of Figures xvii

1 Introduction 1

1.1 Static Verification Of Imperative Programs 2

1.2 Overview of Contributions 4

1.3 Summary 6

I Verification Condition Generation 7

2 Overview 9

3 Phase I: From Source to Intermediate Language 13

3.1 BoogiePL$^\flat$ 14

3.2 Source Language 15

3.2.1 Syntax 16

3.2.2 Typing 18

3.2.3 Operational semantics 20

3.2.4 Modularisation 25

3.3 Translation 27
3.3.1 Logical Part 28
3.3.2 Procedural Part 29

4 Formal Semantics of the Intermediate Verification Language 31
4.1 Syntax 31
4.2 Operational Semantics 32
4.3 Conclusion 39

5 Phase II: Generating Verification Conditions 41
5.1 Hoare Logic 41
5.2 Strongest Postcondition 43
5.3 Weakest Liberal Precondition 46
5.4 Weakest Precondition 49
5.5 Conclusion 56

6 Efficient Weakest Preconditions 57
6.1 Single Assignment Form 59
6.2 Passification 63
6.3 Efficient Weakest Preconditions 64
6.4 Soundness and Size 66
6.5 Conclusion 68

II Symbolic Execution and Separation Logic 71

7 Introduction 73

8 VeriFast Introduction 77
8.1 Separation Logic 77

8.1.1 Rationale . . . 78
8.1.2 Separating Conjunction and Frame Rule . . . 80
8.2 VeriFast: Hands On . . . 82
8.2.1 Basics . . . 82
8.2.2 User Defined Predicates . . . 84
8.2.3 Recursive User Defined Predicates . . . 89
8.2.4 List Reversal . . . 91
8.2.5 Ambiguous Matches . . . 99
8.3 Related Verification Tools . . . 100
8.4 Conclusion . . . 100

9 Featherweight VeriFast 103

9.1 Small Imperative Language . . . 103
9.2 Overview . . . 107
9.3 Result Algebra . . . 108
9.3.1 Examples . . . 108
9.3.2 Operations and Axioms . . . 110
9.3.3 Lemmas . . . 116
9.4 Operators . . . 120
9.5 Basic Operators . . . 126
9.6 Result Algebra Models . . . 130
9.6.1 Inductive Formulae . . . 133
9.6.2 Weakest Preconditions . . . 136
9.7 Operator Lemmas . . . 138
9.8 Concrete Execution . . . 142
9.8.1 Formalisation . . . 142
9.8.2 Shortcomings . . . 149

9.9 Semiconcrete Execution 151

9.9.1 Formalisation 151

9.9.2 Relation with Concrete Execution 159

9.10 Symbolic Execution 173

9.10.1 Formalisation 173

9.10.2 Relation with Semiconcrete Execution 186

9.10.3 Relation with Concrete Execution 196

9.11 Conclusion 197

10 Automation 199

10.1 General Approach And Rationale 200

10.2 Working Example 201

10.3 Auto-open and Auto-close 206

10.4 Autolemmas 207

10.5 Shape Analysis 208

10.6 Comparison 210

10.7 Conclusion 211

11 Conclusion and Future Work 213

11.1 Summary 213

11.2 Verification Condition Generation vs Symbolic Execution 215

11.2.1 Similarities 215

11.2.2 Performance 217

11.3 Related Work 219

11.3.1 Verification Condition Generation 220

11.3.2 Symbolic Execution and Separation Logic 221

11.4 Future Work 224

11.4.1 Further Formalisation of Verification Condition Generation224

11.4.2 Featherweight VeriFast 225
11.4.3 VeriFast 226
11.4.4 Reflection 227

A Notations 233

A.1 Notation definitions 233

B Monads 239

B.1 Basics 239
B.2 The List Monad 240
B.3 The Maybe Monad 242
B.4 Kleisli Triples 243
B.5 Do Notation 244
B.6 The State Monad 245
B.7 Combining the State and Maybe Monads 248
B.8 Combining the State and List monads 249

C (An Attempt At) A Short Introduction to Coq 251

C.1 The Curry-Howard Isomorphism 251
C.2 Curry-Howard in C++ 252
C.2.1 Objects as Witnesses 252
C.2.2 Moving it to Compile-Time 253
C.2.3 Compile-time Checked Indexing 257
C.2.4 Disadvantages 258
C.3 Curry-Howard in Coq 259
C.3.1 Implementing $\leq$ 261
C.3.2 Implementing Lists 262
C.3.3 Implementing List Indexing 263

C.4 Type Hierarchy . . . 264
C.5 Extraction . . . 266
C.6 Clarifications . . . 270
C.6.1 Result Algebra Definitions . . . 270
C.6.2 Result Algebra Axioms . . . 271
C.6.3 Effective Result Algebra . . . 273
C.6.4 Inductive Formulae Model . . . 274

D Coq Scripts 279

D.1 Assertion . . . 280
D.2 AssocList . . . 281
D.3 BooleanExpression . . . 282
D.4 Chunk . . . 284
D.5 ConcreteExecution . . . 285
D.6 DependentProduct . . . 290
D.7 EInductiveFormulae . . . 291
D.8 ERAAxioms . . . 296
D.9 ERADefinitions . . . 298
D.10 ERANotations . . . 299
D.11 ERAOperators . . . 300
D.12 ESymbolicExecution . . . 303
D.13 EWP . . . 311
D.14 EnsembleExt . . . 373
D.15 EqDec . . . 375
D.16 Expression . . . 376
D.17 FakeModel . . . 378
D.18 Formula . . . 380

D.19 Fresh 383
D.20 Heap 384
D.21 Identifier 386
D.22 InductiveFormulae 387
D.23 ListExt 389
D.24 ListSetExt 390
D.25 Misc 391
D.26 Nat 393
D.27 Notations 394
D.28 Predicate 395
D.29 RAAxioms 396
D.30 RADefaultOperators 398
D.31 RADefinitions 399
D.32 RANotations 400
D.33 RAOperatorTheorems 401
D.34 RAOperators 409
D.35 RATheorems 412
D.36 Routine 422
D.37 SIL 423
D.38 SILPP 425
D.39 SMT 429
D.40 SemiconcreteExecution 435
D.41 SetHolder 442
D.42 SetOfSets 443
D.43 Store 445
D.44 Symbol 447
D.45 SymbolicExecution 448

D.46 Term 456
D.47 VCG 458
D.48 WeakestPreconditions 466

Bibliography **471**

E Publications **481**

E.1 Papers at international conferences and symposia, published in full in proceedings 481
E.2 Internal reports 482

List of Figures

2.1 Translations 10

3.1 A Short BoogiePL$^\flat$ Example 16
3.2 Syntax of the Source Language 17
3.3 Three-Layered Typing 18
3.4 Typing Rules 21
3.5 The Step-Over Relation (Part 1) 27
3.6 The Step-Over Relation (Part 2) 28
3.7 Translation of a Method m 29
3.8 Compilation Scheme 30

4.1 Syntax of the Intermediate Verification Language 32
4.2 Single Step Reduction Rules 34
4.3 Translating a while loop 36

6.1 Overview 58
6.2 Single Assignment Transformation Algorithm 62

8.1 Linked Lists Sharing Nodes 80

9.1 Overview of the Different Abstraction Layers 108

9.2 Basic Operators (in Coq Module `RAOperators`) 127
9.3 Formalisation for Example 9.5.4 131
9.4 Computability Requirements 132
9.5 Coq Implementations of Result Algebra Models 133
9.6 Concrete Execution 150
9.7 Semiconcrete Execution 158
9.8 Symbolic Execution 187
9.9 Execution Comparison 188
9.10 Visualisation of the Relation Between the Different Executions . 197

10.1 Annotation Statistics 200
10.2 Finding a Fixed Point 210
10.3 Annotation Line Count Comparison 211

11.1 Overview of Coq Formalisation (approximate numbers) 216
11.2 Merging Execution Paths 218
11.3 Steps for Weakest Preconditions of Unstructured Programs . . . 225

A.1 General Notations 233
A.2 Logical Notations 234
A.3 Set Related Notations 234
A.4 Source Language Notations (Chapter 3) 235
A.5 Intermediate Verification Language Notations 235
A.6 Hoare Triples 235
A.7 Efficient Weakest Precondition Auxiliary Notations 235
A.8 Result Algebra Notations 236
A.9 SIL/SIL^{++} Notations 236
A.10 Execution Notations 237

B.1 RPN Calculator 247
B.2 Nondeterministic RPN Calculator 250

C.1 Coq Metatype Hierarchy 266

Listings

8.1 A Linked List in C 78
8.2 List Reversal in C 78
8.3 Contract for Linked List Reversal C 79
8.4 Calling the List Reversal Function 80
8.5 Cell Struct 85
8.6 Cell Destruction 85
8.7 Elaborately Leaving a Cell Untouched 85
8.8 A User Defined Predicate for Cells 86
8.9 Updated Contract for destroy 86
8.10 Corrected Version of destroy 86
8.11 Memory Safe Triple 87
8.12 Improved Cell Predicate 88
8.13 destroy with Improved Cell Predicate 88
8.14 triple Proven Fully Functional Correct 88
8.15 List Predicate 89
8.16 Destroying Lists 90
8.17 Verified List Reversal (No Full Functional Crrectness) 92
8.18 List Inductive Data Type 93
8.19 Updated List Predicate 93
8.20 Function Contract 93
8.21 Reverse Function 94
8.22 Append Function 95
8.23 Fully Annotated List Reversal Function 96
8.24 AppendAssoc Lemma 97
8.25 AppendNil Lemma 98
8.26 Reverse Lemmas 99
8.27 Ambiguous Match 100
9.1 Result Algebra Signature in Coq 111
9.2 Result Algebra Axioms in Coq 112
9.3 Effective Result Algebra Signature in Coq 133
9.4 Effective Result Algebra Axioms in Coq 134

9.5 Inductive Formulae Model in Coq 135
10.1 Basic Definitions for Integer Carrying Singly Linked Lists 202
10.2 Linked List Lemmas 203
10.3 A VeriFast-Verified List-Copying Algorithm 204
10.4 Destroying a Node 207
B.1 Linq 242
C.1 File Names in C++ 254
C.2 `LEQ` in C++ 255
C.3 `LEQ<X, Y>` in C++ 256
C.4 Proving that $3 \leq 5$ in C++ 257
C.5 A List with Compile-Time Index Checking 258
C.6 Example Usage of a Compile-Time Checked List 258
C.7 A Loop Using Templates 260
C.8 Circumventing C++'s Type System 261
C.9 $\leq$ in Coq 262
C.10 $3 \leq 5$ in Coq 262
C.11 List in Coq 263
C.12 List indexing in Coq (Ltac) 265
C.13 List indexing in Coq (Gallina) 265
C.14 List indexing in Coq (Gallina) 266
C.15 Type of Sorting Function 267
C.16 Type of Sorting Function (Finite Lists) 268
C.17 `or` vs `sumbool` Type 269
C.18 Result Algebra Signature in Coq 270
C.19 Result Algebra Axioms in Coq 272
C.20 Effective Result Algebra Signature in Coq 273
C.21 Effective Result Algebra Axioms in Coq 275
C.22 Inductive Formulae Model in Coq 276

Chapter 1

Introduction

Quis custodiet ipsos custodes?
(Who watches the watchmen?)

Juvenal

Software pervades our world. It has become our entertainment: our music has been digitised, our movies make no profit without special effects, our games are no fun without realtime 3D graphics. Software has also become an indispensable tool: our books are written using text processors and our homes and bridges are constructed using CAD/CAM programs. Software also permeates our social lives: Facebook has, at the time of this writing, almost a billion users.

But software is also present in our cars, our airplanes, our medical devices, our banking terminals, our power plants, our military weapons. And in these cases, we expect our software to be no less than infallible. Sadly, this is not the case and software failures have led to disasters [115, 114] such as fatal overradiation during medical treatments [19, 26, 84] and airplane crashes [85, 74] involving hundreds of deaths.

It is not our intention to scare the reader into moving back into his ancestor's cave. Instead, we wish to reassure the reader: software verification is a very active research area and much progress has been made this last decade. Many different verification techniques exist, and even more tools apply them with great success.

Next to the many good uses listed above to which software is put, it thus also serves to verify software. But since verification software is just as much subject to mistakes as other software, it is of paramount importance that both the algorithms they employ and the implementation of said algorithms are correct.

This brings us to the subject of this thesis. In the following chapters, we will discuss multiple verification techniques. These techniques are well established: some of them are decades old and they form the basis of many existing verification tools.

Our contribution consists of proving these techniques' soundness, i.e. that they produce correct results. While in many cases proofs already existed, we aim for a higher level of rigor: most of our proofs [110] are machine checked. For this, we rely on Coq [33, 16], a proof assistant. This tool aids in the development and checking of mathematical proofs. As far as we know, this can be considered the most rigorous form of proving currently available.

1.1 Static Verification Of Imperative Programs

Let us first define what we understand under "static verification of imperative programs", starting with "imperative programs" for the sake of completeness. In simple terms, we can distinguish two styles of programming: functional and imperative. Both styles use variables to store values. Functional style entails that once a variable has been assigned a value, it is bound to this value forever. In contrast, imperative style allows us to reassign new values to these variables at any moment. Imperative style effectively adds a dimension of time to programs compared to functional programs: the same piece of code can act differently depending on the time at which it is executed.

While functional style is far easier to reason about, in practice, the imperative style is by far the most in use (mostly for reasons of efficiency, modularity and legacy). By choosing imperative programs as the subject of our verification algorithms, we may be taking the difficult path, but it is also the most general path: functional code can be seen as a special subset of imperative code, meaning that all verification algorithms we discuss will also be applicable on functional programs.

The term "verification" refers to the act of ascertaining that a given set of correctness properties hold. There is an entire spectrum of such properties: in languages with manual memory management we might wish to check that there are no memory leaks. In multithreaded programs we might want to make sure that no race conditions or deadlocks can occur. Full functional correctness is one of the most ambitious verification targets, as it not only requires the program not to crash or hang, but also to always produce correct results. Still other programs can benefit of information flow analysis so as to prevent sensitive data from reaching unintended recipients.

The verification techniques we will discuss are not inherently limited to any specific verification property mentioned above. On the contrary: they are generally capable of proving verification properties ranging from memory safety to full functional correctness.

Only "static" remains to be explained, which is easiest by contrast. One way of verifying that a program produces correct results consists of letting it verify its own results. This is often possible as it is generally easier (if not trivial) to verify a result than to compute it. For example, a sorting algorithm could check, prior to returning its result, whether it did successfully accomplish its task. While this "runtime checking" is easy to implement, it suffers from many flaws:

- Runtime checking inflicts a performance penalty: programs run slower and consume more memory.
- What should happen when the program discovers an erroneous result? It could deliberately crash, thereby guaranteeing that *if* it does produce a result, it is correct. In some situations, this might be acceptable, but we would prefer a more robust solution.
- Not every verification property can be checked this way: it is unclear how data races could be detected this way.
- If is possible to forget implementing some runtime checks, which severely undermines trustworthiness. Admittedly, this can be remedied by extending the compiler so that checks are automatically added.
- Even a perfectly runtime checked program can only make limited promises about its behaviour. Say the program forces a crash when a mistake is detected, it is then perfectly possible it will do so after five minutes of usage, or even never start up.

Static verification instead operates before (or during) compile time: it analyses the program's source code and determines whether certain verification properties could be violated at runtime. If the algorithm cannot fully ascertain that no violations will occur during execution, verification fails. Static verification, while more complex to achieve, shares only few of the disadvantages of runtime checking listed above:

- There is no performance penalty, as the code remains unchanged. The converse can even be true. For example, type systems, an ubiquitous kind of static verification, provide the compiler with extra information so as to allow optimisations, thereby enhancing a program's performance.

- Error detection is moved up to compile time, i.e. a program will not compile if it contains bugs. Once the program starts executing, no errors (the ones looked for, at least) can occur.
- Some verification properties may be very difficult to check using static verification.
- Advanced static verification often demands considerable effort from the programmer and requires a specialised skill set.

1.2 Overview of Contributions

This thesis is split into two parts. The first focuses on techniques based on verification condition generation. In short, a verification tool will generate a logical formula from the program's source code whose validity implies the correctness of the program. The actual task of determining a formula's validity is delegated to a (preferably automated) theorem prover, such as Z3 [37], Simplify [39] or Coq [33, 16].

Although the theoretical foundation of this approach is relatively old and more modern developments are available, it is still relied upon by many of today's verifiers such as VCC [31], Dafny [78] or tools built on the Why verification platform [50, 17].

We will discuss multiple verification condition generation algorithms, formalise them and prove their soundness. Definitions, theorems and proofs [110] of this part have also been implemented in the Coq proof assistant [33, 16], which means that all proofs have been machine checked, dramatically increasing our confidence in their correctness.

The second part focuses on symbolic execution [75] and separation logic [98], which constitute a fundamentally different approach to verification than verification condition generation. Separation logic is a relatively recent development in the field of software verification. It provides specialised support for dealing with shared mutable data structures, which are common in imperative programs, and provides an elegant solution to the frame problem. We postpone a more detailed discussion until later.

Many verification tools (e.g. Smallfoot [12], jStar [45]) are based on separation logic, one of which is VeriFast [67]. This tool is under active development at the

KU Leuven and has been used for multiple real world projects [95, 68]. However, it has no solid theoretical foundation, such as for example exists for Smallfoot [4, 108]. This thesis attempts to fill this lacuna: we define Featherweight VeriFast, a small verifier that uses the same core concepts as VeriFast (i.e. separation logic, symbolic execution, heap abstraction, modularisation, etc. all of which will be explained in due time).

First, we will give a full mathematical description of the algorithms underlying Featherweight VeriFast. Next, we will formally prove that together they form a sound verifier. These algorithms have all been implemented in Gallina, Coq's specification language. While it would be possible and desirable to mechanise the soundness proof, time constraints have prevented us from doing so: many lower level lemmas have been proved in Coq, but unfortunately our mechanisation process did not reach the top level theorems.

If certain conditions are met, Coq can perform a program extraction. Put succinctly, this allows us to compile Featherweight VeriFast into an actual executable program. We have taken the necessary measures to satisfy these conditions, meaning the end result of our work is a fully functional sound verifier.

In a next chapter, we discuss automation techniques for separation logic based verifiers. Verifiers can be fully automated, requiring only a push of a button to perform their task. However, these tools can generally only check for a limited set of correctness properties: for example, while verifying memory safety is automatable, full functional correctness remains out of their reach.

Other tools, such as VeriFast, have chosen to rely on the programmer's help, requiring source code to be annotated. This extra effort pays off as this enables the verifier to check for a much larger range of correctness properties.

Unfortunately, the amount of required annotations can be considerable, sometimes doubling or even tripling the size of the source code. To mitigate this, we have developed a number of automation techniques which aim to generate as many annotations as possible, thereby relieving the programmer of some of the burden imposed by manual verification tools.

Lastly, we will propose a verification framework whose purpose it is to allow any number of untrusted automation tools to be used without compromising the soundness of the verification result. It also makes an iterative approach to verification possible, in which verification is performed in multiple steps, the advantages of which we will discuss elaborately.

1.3 Summary

The ubiquity, the growing complexity and our increasing reliance on software has made it necessary to develop tools to assist us in writing correct programs. However, software verification is no trivial task and verification tools are just as much subject to mistakes as other software. For this reason, we have fully formalised a selection of verification algorithms and proven their soundness. The most part has also been machine checked, leading to increased trust. In short, we attempt to provide an answer to the question "who verifies the verifiers?"

Part I

Verification Condition Generation

Chapter 2

Overview

Verification condition (VC) generation is one of the classic techniques for program verification: from the program and its specification one computes a set of logical sentences (the verification conditions) whose validity implies the correctness of the program with respect to the given specification. The technique can be traced back to the very roots of program verification [42].

Example 2.0.1. *As a simple illustration of this concept, consider code which computes the Lorentz factor*[1]*:*

$$\mathit{lorentz}(v) = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

In order to successfully compute[2] *this expression's value, we must put some restrictions on the values v can take:*

- *To prevent complex values from appearing in our result, we need $\frac{v^2}{c^2} \leq 1$.*
- *We strengthen this last condition to $\frac{v^2}{c^2} < 1$ to avert a division by zero.*

Put more succinctly, successful evaluation requires $0 \leq |v| < c$. This condition can thus be seen as a verification condition for computing the Lorentz factor. Since the verification condition is not a tautology, evaluating lorentz(v) *for arbitrary v might go wrong. We can solve this problem by restricting the function's domain: we require*

[1] The Lorentz factor is an important component of the theory of special relativity and expresses among other things by what factor time slows down when travelling at high speeds. However, for our purposes, it is just a mathematical expression.

[2] Here, successful means "yielding a real value ($\mathbb{R}$), assuming v and c to be real."

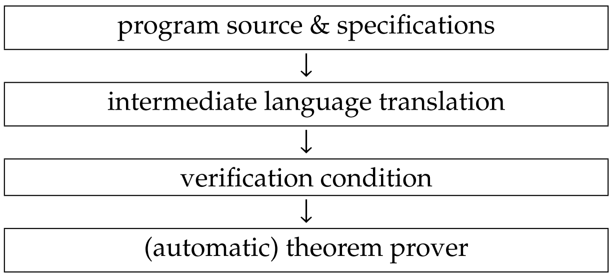
program source & specifications
intermediate language translation
verification condition
(automatic) theorem prover

Figure 2.1: Translations

that $-c < v < c$. We will see later that this restriction corresponds to establishing a precondition *for the function and that* $0 \leq |v| < c$ *is actually a* weakest precondition.

Analogously, we can use verification conditions to prevent out-of-bound array indexing, dereferencing invalid pointers, etc. or, in other words, to ensure that execution stays within the bounds of well-defined behaviour (in the case of C or C++), or does not raise unwanted exceptions (Java, C$^\sharp$, . . .). By adding program specifications (e.g. preconditions, postconditions, invariants) it is also possible to enforce specific behaviour.

However, programming languages have grown more and more complex, supporting features such as dynamic memory allocation, pointers, exception handling mechanisms, objects, inheritance, dynamic binding and so forth, making the process of generating VCs also significantly more complex. To master this increased complexity, many modern program verifiers split the VC generation in two phases. As shown in Fig. 2.1, first the source program and its specification are compiled to an intermediate verification language, and then VCs are generated from the intermediate language. A prominent example of such an intermediate language is the BoogiePL language [82, 7]. BoogiePL is the intermediate language of the Spec$^\sharp$ program verifier [9] and the VCC verifying C compiler [100], and the ESC/Java [51] line of verifiers is moving to a very similar intermediate language.

Another unfortunate consequence of the increased complexity of VC generation for current programming languages is that soundness proofs are either omitted (and hence the VC generation is seen as some form of axiomatic semantics of the source language [82]), or are presented only informally.

The main focus of this part lies on the translation from intermediate language to verification condition. However, to improve understandability, we also provide an elaborate example of the translation from source language to intermediate language, which can be found in Chapter 3.

Chapter 4 defines an intermediate verification language and its operational semantics. Next, in Chapter 5 we discuss multiple verification condition generation algorithms for this intermediate verification language, such as the strongest postcondition and the weakest precondition.

Since the weakest precondition algorithm is the one used most in practice, it deserves a bit more of our attention. The algorithm suffers a major drawback: the generated verification condition grows exponentially with the size of the program. Chapter 6 describes a more efficient algorithm: it can produce verification conditions which grow only polynomially, if the input program satisfies certain criteria. Fortunately, it is possible to transform any program into an equivalent form which satisfies these criteria. We will show that both the program transformation and the polynomial VC generation algorithms are sound.

Our contribution is not the development of the discussed verification generation algorithms, but the full formalisation in Coq accompanied by (machine-checked) proofs of soundness of all presented approaches to VC-generation.

Most of the results in this part of this thesis have been reported in the following publications:

- Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 570–581. Springer, 2009.
- Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 2517–2522. ACM, 2010.

Chapter 3

Phase I: From Source to Intermediate Language

Chapter 2 discusses how verification condition generation can be split up into two phases (see Fig. 2.1) by introducing the use of an intermediate verification language. This chapter focuses on the first phase, i.e. the translation from source language (e.g. Java, C, . . .) to intermediate language.

In this chapter, we use BoogiePL$^\flat$ [82] as the intermediate verification language. It is essentially a stripped down version of BoogiePL [38], the intermediate verification language used by many program verifiers such as VCC [31], HAVOC [76], Dafny [78], Chalice [80] and Spec$^\sharp$ [9]. Section 3.1 gives a short introduction to BoogiePL$^\flat$.

We then proceed to formalise an example object oriented language (Sect. 3.2) by defining its syntax (Sect. 3.2.1), its typing rules (Sect. 3.2.2) and its semantics (Sect. 3.2.3). Lastly, we show how to translate it into BoogiePL$^\flat$(Sect. 3.3).

Since this chapter serves only as an elaborate illustration of how to translate to an intermediate verification language, we have omitted (most) theorems and proofs. A more thorough version of this chapter is available in [111]. Also note that no Coq implementation exists for this part.

3.1 BoogiePL$^\flat$

A BoogiePL$^\flat$ program consists of two parts: on the one hand, a logical part that defines constants, function symbols and axioms. The constants and functions define the value domains of the program and become part of the first-order logical signature in terms of which expressions and assertions are formulated. The signature plus the axioms constitute a classical logical theory. As this part is fairly standard, we refer the interested reader to [82] for a more detailed description. Suffice it to say that it has an axiomatisation for integers, finite maps, booleans and so forth.

On the other hand there is the imperative part that consists of (1) global variables which take values in the mathematical structure axiomatised by the logical part and contribute to the program state, and (2) a number of procedures that can be thought of as describing the possible control-flow paths in the program being verified. For example, as BoogiePL$^\flat$ does not provide a heap, a translation can instead encode it as a global variable being a map of (object reference, field name) pairs to values in the case of an object oriented source language.

Procedures are parameterised operations on the state space defined by the global variables. A procedure's body consists of a single command. The following commands are provided:

- variable declaration, written **var** *id* : *type*, introduces a new variable with unknown initial value.
- **assert** *expression* states that the expression must evaluate to true when execution passes that point, which can be used to specify proof obligations.
- **assume** *expression* tells the verifier that the given expression can be assumed to be true, e.g. preconditions can be assumed to be true at the beginning of a procedure or postconditions to hold just after a procedure call.
- **havoc** *identifier* is the opposite of assume; it removes any information about the specified variable by assigning an arbitrary value to it.
- choice, written c_1 [] c_2, represents a control flow fork: execution could continue with either c_1 or c_2. This command is typically used to model conditional branches, such as if- or while-statements.
- assignment, written x := *expression*, changes the variable x's value.
- sequential composition, written $c_1; c_2$.

- procedure calls *id* := **call** *id*(*Expression**).
- blocks allow grouping commands together.

Since procedure specifications are so common, BoogiePL$^\flat$ supports them directly: one can define procedure specifications followed by one or more procedure implementations, which all have to obey the specifications. The specifications consist of a number of **requires** clauses (the preconditions), **ensures** clauses (the postconditions), and a **modifies** clause, which indicates which global variables have their values changed by the procedure.

Example 3.1.1. *Figure 3.1 shows an example taken from [82]. The first three lines form the logical part. It introduces an integer constant K and a uninterpreted function f whose behaviour is only described by the axiom on the third line, i.e. the only thing we know is that there exists some k such that $f(k) = K$. The procedural part of the code declares a procedure Find which according to its specification sets out to find a k for which $f(k) = K$. A recursive implementation is then given. The body can be seen as an if-then-else with three branches*[1].

3.2 Source Language

Our source language has support for classes which contain fields and methods. There are no constructors: newly created objects (using **new**) have their fields initialised to **null**. There is no support for inheritance. There are no built-in types, meaning there are no integers or booleans, though it is certainly possible to define them.

Methods must be annotated with a specification which consists of a precondition and a postcondition. A precondition must be true at method invocation while conversely a postcondition must be true at method exit.

Preconditions can refer to the method arguments (including the **this** reference), and through these, the heap. We model the preconditions as functions taking two arguments, written $Pre_m(H, F)$ where m is the method whose preconditions we refer to, H represents the heap and F the frame containing the necessary information about the arguments' values.

Postconditions have access to both the "old state" (i.e. the state at method invocation), and the "new state" (i.e. the state at method exit), hence

[1]Technically, in order for it to represent a true if-then-else, the second assume should have condition $f(a) \neq K \wedge f(b) = K$, since if-then-else chains have a clear order in which each condition is considered: an else-clause can assume that the conditions of all preceding tests evaluated to false.

```
const $K$ : int;
function $f$(int) returns (int);
axiom ($\exists\, k$ : int. $f(k) = K$);

procedure $Find(a$ : int, $b$ : int) returns ($k$ : int);
   requires $a \leq b \wedge (\forall\, j$ : int. $a < j \wedge j < b \Rightarrow f(j) \neq K)$;
   ensures $f(k) = K$;
implementation $Find(a$ : int, $b$ : int)
{
   assume $f(a) = K$;
   $k := a$
[]
   assume $f(b) = K$;
   $k := b$
[]
   assume $f(a) \neq K \wedge f(b) \neq K$;
   call $k := Find(a - 1, b + 1)$
}
```

Figure 3.1: A Short BoogiePL$^\flat$ Example

postconditions can refer to the method arguments, the old heap, the new heap, and the method's return value. We model postconditions as functions $Post_m(H, F, H', R)$ with m the method whose postcondition we are referring to, H the old heap, F containing all information about argument values, H' the new heap and R the method's return value.

3.2.1 Syntax

Definition 3.2.1 (syntax)**.** *The syntax of our source language is summarised in Fig. 3.2.*

Note the rather unorthodox syntax for method call. The sole reason for adding an extra type annotation is to simplify the operational semantics. Given a method name m, it is necessary to know the type of the object the method is invoked on as more than one class could contain a method with this name m. Instead of dragging along this typing information in the operational semantics, we have chosen to put this burden on the syntax[2]. Another possible solution to

[2]This syntactic choice forces us to only make use of static dispatch, but this is no restriction as the language does not support subclassing anyway.

Program	::=	*Class**
Class	::=	**class** *Identifier* { *Field** *Method** }
Field	::=	**field** *Identifier* : *Type*
Method	::=	**method** *Identifier* (*Parameter**) : *Type Specification* { *Statement** }
Argument	::=	*Identifier* : *Type*
Specification	::=	**requires** *Precondition* **ensures** *Postcondition*
Statement	::=	**local** *Identifier* : *Type* ;
	|	*Identifier* = **null** ;
	|	*Identifier* = *Identifier* . *Identifier* ;
	|	*Identifier* = *Identifier* . *Type* :: *Identifier* (*Identifier**) ;
	|	*Identifier* = **new** *Identifier* ;
	|	*Identifier.Identifier* = *Identifier* ;
	|	**if** (*Identifier* == *Identifier*) *Statement* **else** *Statement*
Type	::=	*Identifier*

Figure 3.2: Syntax of the Source Language

this problem would be to forbid a method name to be used in more than one class.

The language is intentionally kept minimal to keep the formalisation simple, but it is still Turing complete. For example, the **if** statement may seem overly specific in that it does not allow us to compare x.f with y.g, but using temporary variables as follows:

local t1 : C1; **local** t2 : C2; t1 = x.f; t2 = y.f; **if** (t1 == t2) . . .

Also worthy of attention is the lack of a **return** statement. We took the same approach as the Eiffel-language: a special local variable named **result** (initialised to **null**) contains the return value of the method. One can image there being a **return result**; statement at the end of each method body.

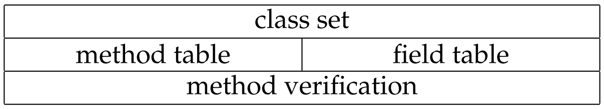

Figure 3.3: Three-Layered Typing

3.2.2 Typing

Typing happens in three layers (see Fig. 3.3), every layer only depending on information accumulated in the one before it. To type-check a program P we follow the following steps:

- First, we collect all class names in what we call P's class set (see Def. 3.2.2).
- Next, we build a method table (Def. 3.2.4) containing type information about the methods (i.e. argument types, return type) and field table (Def. 3.2.5) which stores the fields' types. Both these tables are well-formed (Def. 3.2.6 and Def. 3.2.7) if all types are contained in the class set.
- Lastly, we check each method body in turn, making sure every statement is valid.

Definition 3.2.2 (class set)**.** *Given a program P, we define the* class set Δ_c^P *as the set containing the names of all defined classes in the program.*

$$\textbf{class}\ C\ \{\ldots\} \in P \iff C \in \Delta_c^P$$

Definition 3.2.3 (class body)**.** *We define the* class body *of C, written* classbody(C) *as the set of method and field declarations contained in the class declaration.*

Definition 3.2.4 (method table)**.** *Given a program P, the* method table Δ_m^P *is a partial map from class names to a partial map from names to method types.*

$$\Delta_m^P(C)(id) = C_1 \to C_2 \to \cdots \to C_n \to C_r$$

where

$$\textbf{method}\ id(x_1 : C_1,\, x_2 : C_2,\, \ldots,\, x_n : C_n) : C_r \ldots \in \text{classbody}(C)$$

Definition 3.2.5 (field table)**.** *Given a program P, the* field table Δ_f^P *is a partial map from class names to a partial map from names to field types.*

$$\Delta_f^P(C)(id) = C_f$$

where

$$\textbf{field}\ id\!:\!C_f \ \in\ \text{classbody}(C)$$

Definition 3.2.6 (well-formed method table). *We say a program P's method table Δ^P_m is* well-formed, *written Δ^P_m* ok, *if every type mentioned in it is an element of P's class set Δ^P_c.*

$$\Delta^P_m \text{ ok} \iff \forall\, C, id.\ \left(\begin{array}{c} \Delta^P_m(C)(id) = C_1 \to C_2 \to \cdots \to C_n \to C_r \\ \Downarrow \\ C_1 \in \Delta^P_c \ \wedge\ C_2 \in \Delta^P_c \ \wedge\ \ldots\ \wedge\ C_n \in \Delta^P_c \ \wedge\ C_r \in \Delta^P_c \end{array}\right)$$

Definition 3.2.7 (well-formed field table). *We say a program P's field table Δ^P_f is* well-formed, *written Δ^P_f* ok, *if every type mentioned in it is an element of P's class set Δ^P_c.*

$$\Delta^P_f \text{ ok} \iff (\forall\, C, id.\ \Delta^P_f(C)(id) = C_f \Rightarrow C_f \in \Delta^P_c)$$

Definition 3.2.8 (program data). *Given a program P, P's* program data, *written Δ^P is the triplet $(\Delta^P_c, \Delta^P_m, \Delta^P_f)$.*

Definition 3.2.9 (well-typed method). *Given a program P, a method m from a class C is* well-typed *if*

$$\Delta^P; \Gamma \vdash \bar{s} \text{ ok}$$

where

$$\begin{array}{rcl} \textbf{method}\ m(x_1{:}C_1, \ldots, x_n{:}C_n){:}C_r\ specs\ \{\bar{s}\} & \in & \text{classbody}(C) \\ \Gamma & = & \epsilon, x_1 : C_1, \ldots, x_n : C_n, \textbf{result} : C_r \end{array}$$

Figure 3.4 contains the actual typing rules.

The typing rules are standard and kept simple mainly due to the lack of subtyping. We briefly clarify the typing rules:

- T-Local requires a newly declared local variable x to be fresh with respect to the current environment, i.e. it is not allowed to reuse identifiers in the same scope. The type C of x must be a declared class, and the rest of the program $\bar{s}$ must be well-typed given that x has type C.
- The typing rules (more specifically T-StoreNull) don't make any special demands regarding the assignment of **null** to variables.
- T-StoreField deals with the $x{=}y.f$ command and requires that y is of some type C_y which declares a field f which must have the same type as x. The type rule T-WriteField makes similar demands.

- T-StoreNew requires x to have been declared of type C if it is to be assigned a new object of type C.
- T-If requires x and y to have the same type.
- T-MethodCall demands that y is of a type C which declares a method named m. All arguments z_i must have types that match the parameter types C_i. The method's return type C_r must match x's declared type.

Definition 3.2.10 (well-typed program)**.** *A program is* well-typed *if its method and field tables are well-formed and if every method of every class is well-typed.*

From now on, we will always assume every piece of code we deal with is well-typed.

3.2.3 Operational semantics

In this section we define an operational semantics for our source language. The operational semantics is modelled as a binary relation on program states. Thus, we first define the program states, after which we define both the single step $\leadsto$ and multiple step $\leadsto^*$ reduction rules.

State

We define the operational semantics as a binary relation on states, written $\sigma_1 \leadsto \sigma_2$. Operational semantics describe how execution proceeds step by step: $\sigma_1 \leadsto \sigma_2$ expresses that execution a single step while in state σ_1 leads to a new program state σ_2. Chaining these together $\sigma_1 \leadsto \sigma_2 \leadsto \sigma_3 \leadsto \ldots \leadsto \sigma_n$ represents the entire execution of the program.

We distinguish two kinds of states: a failure state, written Fail, and an in-progress state which is a quintuplet containing

1. The heap: a partial function mapping object identifiers (*oid*s) to objects, which themselves are partial functions mapping field names to *oid*s.
2. The store: a stack of frames, which are partial functions mapping identifiers to *oid*s. A frame keeps track of the local variable bindings. At each method invocation, a new frame is pushed onto the store and conversely, at each method exit, a frame is popped from the store.

$$\frac{x \notin \mathrm{dom}(\Gamma) \quad C \in \Delta^P_c \quad \Delta^P;(\Gamma, x : C) \vdash \bar{s} \text{ ok}}{\Delta^P;\Gamma \vdash \textbf{local } x{:}C;\, \bar{s} \text{ ok}} \text{ T-LOCAL}$$

$$\frac{\Delta^P;\Gamma \vdash \bar{s} \text{ ok}}{\Delta^P;\Gamma \vdash x{=}\textbf{null};\, \bar{s} \text{ ok}} \text{ T-STORENULL}$$

$$\frac{x : C_x \in \Gamma \quad y : C_y \in \Gamma \quad C_x = \Delta^P_f(C_y)(f) \quad \Delta^P;\Gamma \vdash \bar{s} \text{ ok}}{\Delta^P;\Gamma \vdash x{=}y.f;\, \bar{s} \text{ ok}} \text{ T-STOREFIELD}$$

$$\frac{x : C \in \Gamma \quad \Delta^P;\Gamma \vdash \bar{s} \text{ ok}}{\Delta^P;\Gamma \vdash x{=}\textbf{new } C;\, \bar{s} \text{ ok}} \text{ T-STORENEW}$$

$$\frac{x : C_x \in \Gamma \quad y : C_y \in \Gamma \quad C_y = \Delta^P_f(C_x)(f) \quad \Delta^P;\Gamma \vdash \bar{s} \text{ ok}}{\Delta^P;\Gamma \vdash x.f{=}y;\, \bar{s} \text{ ok}} \text{ T-WRITEFIELD}$$

$$\frac{x : C \in \Gamma \quad y : C \in \Gamma \quad \Delta^P;\Gamma \vdash s_1 \text{ ok} \quad \Delta^P;\Gamma \vdash s_2 \text{ ok} \quad \Delta^P;\Gamma \vdash \bar{s} \text{ ok}}{\Delta^P;\Gamma \vdash \textbf{if } (\ x == y\ )\ s_1 \textbf{ else } s_2;\ \bar{s} \text{ ok}} \text{ T-IF}$$

$$\frac{\begin{array}{c} x : C_r \in \Gamma \qquad y : C \in \Gamma \qquad z_1 : C_1 \in \Gamma\ \ldots\ z_n : C_n \in \Gamma \\ \Delta^P_m(C)(m) = C_1 \to \cdots \to C_n \to C_r \qquad \Delta^P;\Gamma \vdash \bar{s} \text{ ok} \end{array}}{\Delta^P;\Gamma \vdash x{=}y.C{::}m(z_1,\ldots,z_n);\, \bar{s} \text{ ok}} \text{ T-METHODCALL}$$

$$\frac{}{\Delta^P;\Gamma \vdash \epsilon \text{ ok}} \text{ T-NIL}$$

Figure 3.4: Typing Rules

3. The condition stack: it is used to keep track of methods' postconditions. At each method invocation, the postcondition is pushed onto this stack, and at method exit, it is popped and verified.
4. The receiver stack. A receiver is a local variable to which the result value of a method invocation needs to be written to, e.g.

   x = y.T::m();

   In this example, x is the receiver. The receiver stack has the same dynamics as the previously mentioned once: a new item is pushed on method invocation and one is popped at method exit.
5. A statement stack: this stack contains statement lists, each containing (part of) a method body.

Reduction rules

In this section we define the operational semantics as both small step $\rightsquigarrow$ and multiple step relations $\rightsquigarrow^*$. Neither have any surprising elements: they are deliberately kept as straightforward as possible.

Definition 3.2.11 (single step)**.** *We define the following* single step *reduction rules:*

- E-Local*: introduces a new local variable which is initialised to* **null***.*

$$\frac{F' = (F, x \mapsto \mathbf{null})}{(H, F \circ Fs, Cs, Rs, (\mathbf{local}\ x{:}T;\ \overline{s}) \circ Ps) \rightsquigarrow (H, F' \circ Fs, Cs, Rs, \overline{s} \circ Ps)}$$

- E-StoreNull*: writes* **null** *to a local variable.*

$$\frac{F' = (F, x \mapsto \mathbf{null})}{(H, F \circ Fs, Cs, Rs, (x{=}\mathbf{null};\ \overline{s}) \circ Ps) \rightsquigarrow (H, F' \circ Fs, Cs, Rs, \overline{s} \circ Ps)}$$

- E-StoreField*: fetches a field value and stores it in a local variable.*

$$\frac{oid = F[y] \qquad oid \neq \mathbf{null} \qquad F' = (F, x \mapsto H[oid][f])}{(H, F \circ Fs, Cs, Rs, (x{=}y.f;\ \overline{s}) \circ Ps) \rightsquigarrow (H, F' \circ Fs, Cs, Rs, \overline{s} \circ Ps)}$$

- E-StoreFieldNull*: reading a field through a* **null** *reference results in failure.*

$$\frac{F[y] = \mathbf{null}}{(H, F \circ Fs, Cs, Rs, (x{=}y.f;\ \overline{s}) \circ Ps) \rightsquigarrow \textsc{Fail}}$$

- E-StoreNew: *creates a new object and stores a reference to it in a local variable.*

$$\frac{oid \notin H \qquad H' = H, oid \mapsto \mathbb{F} \qquad \mathbb{F} = \epsilon, \overline{f} \mapsto \textbf{null} \qquad F' = (F, x \mapsto oid)}{(H, F \circ Fs, Cs, Rs, (x\text{=}\textbf{new}\ K; \overline{s}) \circ Ps) \rightsquigarrow (H', F' \circ Fs, Cs, Rs, \overline{s} \circ Ps)}$$

- E-WriteField: *writes a value to an object field.*

$$\frac{\begin{array}{c} oid = F[y] \neq \textbf{null} \qquad \mathbb{F} = H[oid] \qquad v = F[y] \\ \mathbb{F}' = \mathbb{F}, f \mapsto v \qquad H' = H, oid \mapsto \mathbb{F}' \end{array}}{(H, F \circ Fs, Cs, Rs, (x.f = y; \overline{s}) \circ Ps) \rightsquigarrow (H', F \circ Fs, Cs, Rs, \overline{s} \circ Ps)}$$

- E-WriteFieldNull: *attempts to write to a field through a* **null** *reference results in failure.*

$$\frac{F[x] = \textbf{null}}{(H, F \circ Fs, Cs, Rs, x.f = y; \overline{s} \circ Ps) \rightsquigarrow \textsc{Fail}}$$

- E-IfTrue: *if the condition is true, the* **then***-branch must be executed next.*

$$\frac{F[x] = F[y]}{(H, F \circ Fs, Cs, Rs, (\textbf{if}\ (x == y)\ s_1\ \textbf{else}\ s_2; \overline{s}) \circ Ps) \rightsquigarrow (H, F \circ Fs, Cs, Rs, (s_1; \overline{s}) \circ Ps)}$$

- E-IfFalse: *if the condition is false, the* **else** *branch must be executed next.*

$$\frac{F[x] \neq F[y]}{(H, F \circ Fs, Cs, Rs, (\textbf{if}\ (x == y)\ s_1\ \textbf{else}\ s_2; \overline{s}) \circ Ps) \rightsquigarrow (H, F \circ Fs, Cs, Rs, (s_2; \overline{s}) \circ Ps)}$$

- E-MethodCall: *method invocation $x\text{=}y.T\text{::}m(\overline{z})$. In case that y points to a valid object (i.e. not* **null***) and the precondition is satisfied by the current state, the new body is added to the statement stack, a new stack frame is created containing bindings for the* **this** *reference, method arguments and* **result** *variable. The method's postcondition is pushed on the condition stack with the before-state already filled in for use later on: upon method exit (described by* E-ExitMethod *and* E-ExitMethodFail*) the postcondition will be popped from the condition stack and evaluated. Lastly, the receiver variable (the one to which the method result will be written to upon return) is pushed on the receiver stack.*

$$\frac{\begin{array}{rcl} F' & = & \epsilon, \textbf{this} \mapsto oid, \overline{args_{T::m}} \mapsto F[\overline{z}], \textbf{result} \mapsto \textbf{null} \\ oid & = & F[y] \\ oid & \neq & \textbf{null} \\ Pre_{T::m}(H, F') & = & true \end{array}}{\begin{pmatrix} H \\ F \circ Fs \\ Cs \\ Rs \\ (x\text{=}y.T\text{::}m(\overline{z});\ \overline{s}) \circ Ps \end{pmatrix} \rightsquigarrow \begin{pmatrix} H \\ F' \circ F \circ Fs \\ Post_{T::m}(H, F') \circ Cs \\ x \circ Rs \\ body_{T::m} \circ \overline{s} \circ Ps \end{pmatrix}}$$

- E-MethodCallFail: *calling a method whose preconditions aren't satisfied yields failure.*

$$\frac{\begin{array}{rcl} F' & = & \epsilon, \mathbf{this} \mapsto oid, \overline{args_{T::m}} \mapsto F[\overline{z}], \mathbf{result} \mapsto \mathbf{null} \\ oid & = & F[y] \\ oid & \neq & \mathbf{null} \\ Pre_{T::m}(H, F') & \neq & true \end{array}}{(H, F \circ Fs, Cs, Rs, x{=}y.T::m(\overline{z})\ \overline{s} \circ Ps) \rightsquigarrow \textsc{Fail}}$$

- E-MethodCallNull: *invoking a method through a* **null** *reference leads to failure.*

$$\frac{F[y] = \mathbf{null}}{(H, F \circ Fs, Cs, Rs, x{=}y.T::m(\overline{z})\ \overline{s} \circ Ps) \rightsquigarrow \textsc{Fail}}$$

- E-ExitMethod: *pops the necessary stacks and writes the* **result** *value to the receiver variable.*

$$\frac{C(H, F'[\mathbf{result}]) = true \qquad F'' = (F, x \mapsto F'[\mathbf{result}])}{(H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps) \rightsquigarrow (H, F'' \circ Fs, Cs, Rs, Ps)}$$

- E-ExitMethodFail: *exiting a method fails if the method's postcondition isn't satisfied.*

$$\frac{C(H, F[\mathbf{result}]) \neq true}{(H, F \circ Fs, C \circ Cs, Rs, \epsilon \circ \overline{s} \circ Ps) \rightsquigarrow \textsc{Fail}}$$

The reduction rules show in which way a failure state can be reached. We summarise them here.

- E-StoreNull: reading a field through a **null** reference, for example

  x = **null**; y = x.field;

  will inevitably lead to failure.

- E-StoreFieldNull: writing to a field through a **null** reference, for example

  x = **null**; x.f = y;

- E-MethodCallNull: calling a method on a null reference, for example

  x = **null**; y = x.m();

- E-MethodCallFail: calling a method while its preconditions are not satisfied.
- E-ExitMethodFail: returning from a method while its postconditions are not satisfied.

These are the failures we wish to prevent using verification condition generation. If a program verifies, it means that none of these failures will be encountered at runtime.

Definition 3.2.12 (multiple step)**.** *We define a binary* multiple step *relation as follows:*

$$\frac{}{\sigma \rightsquigarrow^* \sigma}\ \text{E}^*\text{-Reflexive}$$

$$\frac{\sigma_1 \rightsquigarrow \sigma_2 \quad \sigma_2 \rightsquigarrow^* \sigma_3}{\sigma_1 \rightsquigarrow^* \sigma_3}\ \text{E}^*\text{-Step}$$

Definition 3.2.13 (initial state)**.** *Given a program P where some class X is the only to contain a nullary preconditionless T-returning method named main, the* initial state *$\sigma_{\text{init}}(P)$ is equal to*

$$(\epsilon, \epsilon \circ \epsilon, \epsilon, \epsilon, \bar{s} \circ \epsilon)$$

where

$$\begin{array}{rcl} \bar{s} & = & \textbf{local}\ x{:}X; \\ & & x = \textbf{new}\ X; \\ & & \textbf{local}\ r : T; \\ & & r = x.X{::}main(); \end{array}$$

3.2.4 Modularisation

During verification, we are not interested in the actual evaluation of the program, but only in whether or not failure is encountered. For this, we split the program up in pieces, i.e. we consider each method apart.

To achieve this, we take a method from the program, we evaluate this method (keeping in mind we only want to detect failures, so we cut down as much as possible), and check that no failures occur. If we do this for every method in the program and find that no single method fails, the program in its entirety won't fail either. In order to do this, we need to apply some changes:

- We need a new kind of state: as we are confining execution to within a method, we don't need the stacks: only the top items are of interest. Hence, a state $(H, F \circ Fs, C \circ Cs, Rs, \bar{s} \circ Ps)$ gets "flattened" to $(H, F, C, \bar{s})$.

- We also need to know what initial state to use. The only guarantee we have is that the method's precondition will be satisfied, so we have to consider every state which satisfies those as initial state.
- A method body often contains calls to other methods. Since we want to limit ourselves to only one method, we must find a way to deal with those invocations. For this, we completely rely on the called method's specifications. Method invocation then gets dealt with by a new reduction rule, which first checks that the current state satisfies the invoked method's preconditions and then produces *a* new state which is only guaranteed to satisfy the method's postconditions, meaning this new method-invocation rule is nondeterministic.
- Not only do we want a method execution not to fail when starting off from a random state satisfying the method's preconditions, we also want it to end in a state which satisfies the method's postcondition. In our operational semantics, **E-ExitMethod** takes care of verifying this. However, we can't use this rule directly as it makes execution leave the current method.

We have given these new rules the name "step over relation", and they are fully defined in Fig. 3.5 and Fig. 3.6. Most H- rules are directly derived from a corresponding E- rule (Def. 3.2.11): in most cases, the only change needed is the replacement of the stack state components by their respective top element.

Definition 3.2.14 (step over relation)**.** *We introduce a new binary relation over states which we call the* step-over relation*. The exact rules are contained in Fig.* 3.5 *and Fig.* 3.6*.*

Definition 3.2.15 (multistep-over relation)**.** *The* multisteps-over *relation is defined as follows:*

$$\frac{}{\sigma \curvearrowright^{*} \sigma}\ \text{H}^{*}\text{-}\textsc{Reflexive}$$

$$\frac{\sigma_1 \curvearrowright \sigma_2 \quad \sigma_2 \curvearrowright^{*} \sigma_3}{\sigma_1 \curvearrowright^{*} \sigma_3}\ \text{H}^{*}\text{-}\textsc{StepOver}$$

Theorem 3.2.1. *If we consider the execution of a method body from immediately after* E-MethodCall *to just before* E-ExitMethod *and replace* E-MethodCall *by a nondeterministic rule that leads to a random state satisfying the invoked method's postcondition, then, if the step rules lead to failure, so will the step-over rules.*

Proof. We refer the interested reader to [111]. □

$$\frac{}{(H, F, C, \mathbf{local}\ x{:}T;\ \overline{s}) \curvearrowright (H, (F, x \mapsto \mathbf{null}), C, \overline{s})}\ \text{H-Local}$$

$$\frac{}{(H, F, C, x{=}\mathbf{null};\ \overline{s}) \curvearrowright (H, (F, x \mapsto \mathbf{null}), C, \overline{s})}\ \text{H-StoreNull}$$

$$\frac{F[y] \neq \mathbf{null}}{(H, F, C, x{=}y.f;\ \overline{s}) \curvearrowright (H, (F, x \mapsto H[F[y]][f]), C, \overline{s})}\ \text{H-StoreField}$$

$$\frac{F[y] = \mathbf{null}}{(H, F, C, x{=}y.f;\ \overline{s}) \curvearrowright \text{Fail}}\ \text{H-StoreFieldNull}$$

$$\frac{oid \notin H \qquad H' = H, oid \mapsto \mathbb{F} \qquad \mathbb{F} = \epsilon, \overline{f} \mapsto \mathbf{null}}{(H, F, C, x{=}\mathbf{new}\ K;\ \overline{s}) \curvearrowright (H', (F, x \mapsto oid), C, \overline{s})}\ \text{H-StoreNew}$$

$$\frac{oid = F[x] \neq \mathbf{null} \qquad \mathbb{F}' = H[oid], f \mapsto F[y]}{(H, F, C, x.f = y;\ \overline{s}) \curvearrowright ((H, oid \mapsto \mathbb{F}'), F, C, \overline{s})}\ \text{H-WriteField}$$

$$\frac{F[y] = \mathbf{null}}{(H, F, C, x.f = y;\ \overline{s}) \curvearrowright \text{Fail}}\ \text{H-WriteFieldNull}$$

$$\frac{F[x] = F[y]}{(H, F, C, \mathbf{if}\ (\ x == y\ )\ s_1\ \mathbf{else}\ s_2;\ \overline{s}) \curvearrowright (H, F, C, s_1\ \overline{s})}\ \text{H-IfTrue}$$

$$\frac{F[x] \neq F[y]}{(H, F, C, \mathbf{if}\ (\ x == y\ )\ s_1\ \mathbf{else}\ s_2;\ \overline{s}) \curvearrowright (H, F, C, s_2\ \overline{s})}\ \text{H-IfFalse}$$

Figure 3.5: The Step-Over Relation (Part 1)

3.3 Translation

Now that we have formally defined our source language, we are ready to translate it to BoogiePL$^\flat$. In Sect. 3.3.1, we first define some axioms which

$$\frac{F' = \epsilon, \textbf{this} \mapsto oid, \overline{args_{T::m}} \mapsto F[\overline{z}] \qquad oid = F[y] \neq \textbf{null} \quad Pre_{T::m}(H,F') = true \qquad Post_{T::m}(H,F')(H',F'') = true}{(H,F,C,x{=}y.T{::}m(\overline{z});\ \overline{s}) \curvearrowright (H',F,C,\overline{s})} \text{ H-MethodCall}$$

$$\frac{F' = \epsilon, \textbf{this} \mapsto oid, \overline{args_{T::m}} \mapsto F[\overline{z}] \quad Pre_{T::m}(H,F') \neq true}{(H,F,C,x{=}y.T{::}m(\overline{z});\ \overline{s}) \curvearrowright \textsc{Fail}} \text{ H-MethodCallPreFail}$$

$$\frac{F[y] = \textbf{null}}{(H,F,C,x{=}y.T{::}m(\overline{z});\ \overline{s}) \curvearrowright \textsc{Fail}} \text{ H-MethodCallNull}$$

$$\frac{C(H,F) \neq true}{(H,F,C,\epsilon) \curvearrowright \textsc{Fail}} \text{ H-MethodExitFail}$$

Figure 3.6: The Step-Over Relation (Part 2)

describe the heap. Next, Sect. 3.3.2 shows how to translate each method of our source language into BoogiePL$^\flat$ commands. This section is mostly inspired by [82], which we refer to reader to for more elaborate examples.

3.3.1 Logical Part

Our verifier promises that if a program verifies, it will never encounter failure during its execution. It does however not make any guarantees regarding a program's success, i.e. if a program does not verify, it means the program *could* fail, but might as well run correctly. In short, false positives (the program verifies) are excluded, but false negatives are not. To maximise a verifier's usability, we wish to minimise the number of false negatives. For this, we can provide BoogiePL$^\flat$ with more information about the heap in the form of axioms.

Definition 3.3.1 (well-formed heap)**.** *We say the heap is* well-formed*, written* $\mathsf{wf_h}(H)$ *if for each object it contains the fields are either* **null** *or refer to other existent objects.*

$$\mathsf{wf_h}(H) \iff (\forall\, oid, oid', f.\ oid' = H[oid][f] \Rightarrow oid' = \textbf{null} \vee oid' \in \mathrm{dom}(H))$$

Theorem 3.3.1. *The multistep preserves the well-formedness of heaps.*

$$\mathsf{wf}_\mathrm{h}(H) \Rightarrow (H, Fs, Cs, Rs, Ps) \rightsquigarrow^* (H', Fs', Cs', Rs', Ps') \Rightarrow \mathsf{wf}_\mathrm{h}(H')$$

Definition 3.3.2 (successor heap)**.** *We say a heap H'* succeeds *a heap H iff*

$$\mathrm{successor}(H, H') \iff \mathsf{wf}_\mathrm{h}(H') \wedge \forall\, oid.\ oid \in \mathrm{dom}(H) \Rightarrow oid \in \mathrm{dom}(H')$$

Theorem 3.3.2. *If H is well-formed, and*

$$(H, Fs, Cs, Rs, Ps) \rightsquigarrow^* (H', Fs', Cs', Rs', Ps')$$

then H' succeeds H.

3.3.2 Procedural Part

We translate each method in turn. Figure 3.7 shows the skeleton, which sets up the environment in which the method will execute: Locals and parameters are declared, the heap is initialised, and the program state is assumed to satisfy the method precondition. Next comes the translation of the actual method body (see Fig. 3.8). Translation ends with an assertion of the method postcondition.

```
var $id_{local}$ : ref;      for every declared local variable in the method
var initheap : [ref, name]any;
initheap := heap;
 var $id_{arg}$;                 }
 var $init\$id_{arg}$;           }   for every argument
 $init\$id_{arg}$ := $id_{arg}$;  }
assume this ≠ null;
assume $\mathsf{wf}_\mathrm{h}$(heap);
assume $Pre_{T::m}$;
Translation of the method body
assert $Post_{T::m}$
```

Figure 3.7: Translation of a Method m

Theorem 3.3.3 (translation soundness)**.** *Given a (well-typed) program P, m a method of P, c its translation to BoogiePL$^\flat$(Fig. 3.7 and Fig. 3.8). If for any H, F for which $Pre_m(H, F)$ the following is true*

$$(H, F, Post_m(H, F), body_m) \curvearrowright^* \textsc{Fail}$$

local x : T;	x := **null**
x = **null**;	x := **null**
x = y.f;	**assert** y ≠ **null**; x := heap[y, f]

x = y.m($\overline{\text{z}}$)	{
	  var oldheap : [name, ref]any;
	  oldheap := heap;
	  assert y ≠ **null**;
	  assert Pre_m;
	  havoc heap;
	  assume successor(oldheap, heap);
	  assume $Post_m$
	}

x = **new** T;	{
	  var oid : **ref**;
	  assume oid ≠ **null**;
	  assume heap[oid, alloc] = **false**;
	  assume (∀ f : **name**. heap[oid, f] = **null**);
	  heap[oid, alloc] := **true**
	}

x.f = y;	**assert** x ≠ **null**; heap[x, f] := y

if (x == y) s$_1$ **else** s$_2$	{
	  assume x = y;
	  translation of s_1
	[]
	  assume x ≠ y;
	  translation of s_2
	}

Figure 3.8: Compilation Scheme

then, for some store μ:

$$\exists\, \mu'.\ \langle c, \mu\rangle \longrightarrow^* \textsf{failure}\ \mu'$$

where $\longrightarrow^*$ refers to BoogiePL$^\flat$'s operational semantics, see next chapter for more details.

Chapter 4

Formal Semantics of the Intermediate Verification Language

This chapter formally defines the intermediate verification language for which we will define verification condition generation algorithms in the next chapter. We have taken our inspiration from BoogiePL$^\flat$, an existing intermediate verification language [82].

The intermediate verification language's syntax is defined in Sect. 4.1, after which Sect. 4.2 formalises its semantics. We also state some theorems which will prove useful in later chapters. Corresponding Coq definitions (see Appendix D) are referred to between parenthesis using `this font`.

4.1 Syntax

In this section we define the syntax of the intermediate verification language (IVL), which consists of the core part of BoogiePL$^\flat$. More specifically, the IVL is a subset of BoogiePL$^\flat$'s command language [82] (see Sect. 3.1 on page 14) which is used to abstractly model the computations to be verified.

Definition 4.1.1 (`VCG.command`). *The syntax is shown in Fig. 4.1.*

We summarise the differences:

command	::=	**assert** *expression*	*assertion*
	|	**assume** *expression*	*assumption*
	|	**havoc** *identifier*	*forgetting*
	|	**skip**	*no-op*
	|	*identifier* := *expression*	*assignment*
	|	*command* [] *command*	*nondeterministic choice*
	|	*command*; *command*	*sequencing*

Figure 4.1: Syntax of the Intermediate Verification Language

- We omitted a procedure call command as it can be desugared to asserting the precondition, havocing the appropriate variables and assuming the postconditions. Details can be found in [82].
- There are no variable declarations in our intermediate verification language; all variables are bound by the store at all times.
- We do not support blocks, which allow us to limit variable scope. These are not strictly necessary as it is possible to move all variables to the outermost scope after having given them unique names.
- We keep our expressions abstract: they are modelled as functions mapping stores to values. There is no need to define a syntax for expressions.
- We added a **skip** command, which, although it amounts to a no-op, considerably simplifies the operational semantics.

The meaning of these commands will be made clear in the next section, where we formally describe the behaviour of each.

4.2 Operational Semantics

In this section, we formalise the semantics of programs written in our intermediate verification language. We do so using small step operational semantics i.e. a binary relation over program states which describes how programs are executed. An execution step transforming an input state σ into an output state σ' is written $\sigma \longrightarrow \sigma'$. σ' is also called σ's *successor*. Full execution then consists of a chain of program states: $\sigma_0 \longrightarrow \sigma_1 \longrightarrow \sigma_2 \longrightarrow \ldots$. Execution ends when it reaches an irreducible state, i.e. a state for which no successor is defined.

Two kinds of program states exist: in-progress states and failure states. The latter are irreducible and indicate that execution has failed. An in-progress state must contain all information needed to execute the program further. Hence, it needs some sort of program counter indicating what commands are left to execute. We achieve this by rewriting the program at every step, i.e. the program state has a command component representing what is left to execute.

The intermediate verification language also supports variables, hence it is necessary to keep track of their bindings. This responsibility falls on the *store*, which maps variables to values. Since we are not interested in performing actual computations, values are kept abstract in our formalisation. The only value of interest to us is true, which is needed to define the semantics of the **assert** and **assume** commands. We now formally define each concept in turn.

Definition 4.2.1 (`VCG.value`, `.value_eq_dec`)**.** *The set of* values *is denoted Val. There exists a value* true $\in$ *Val. The only operation defined on values is the equality test.*

Definition 4.2.2 (`VCG.store`)**.** *A* store μ *is a total function from identifiers to values.*

$$store \;\equiv\; Id \rightarrow Val$$

Since the store is a total function, every variable is bound and one does not have to worry about undeclared variables.

Like values, expressions are kept as abstract as possible. An expression can be an arbitrary mathematical expression. Since it can refer to variables, evaluation requires a store.

Definition 4.2.3 (`VCG.expression`)**.** *An* expression *e is a function from stores to values.*

$$expression \;\equiv\; store \rightarrow Val$$

Example 4.2.1. *The expression* $5 \cdot x + y$ *is modelled by the following function:*

$$\lambda\ \mu.\ 5 \cdot \mu(x) + \mu(y)$$

Definition 4.2.4 (`VCG.state`)**.** *We distinguish two kinds of program states:*

- In-progress states $\langle c, \mu \rangle$ *consist of a command c and a store* μ*.*
- Failure states failure μ *have a store component*[1] μ*.*

We use the symbol σ *to denote states.*

$$\frac{e(\mu) = \text{true}}{\langle \textbf{assert}\ e, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu\rangle}\ \text{E-AssertTrue}$$

$$\frac{e(\mu) \neq \text{true}}{\langle \textbf{assert}\ e, \mu\rangle \longrightarrow \textsf{failure}\ \mu}\ \text{E-AssertFalse}$$

$$\frac{e(\mu) = \text{true}}{\langle \textbf{assume}\ e, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu\rangle}\ \text{E-AssumeTrue}$$

$$\frac{\langle c_1, \mu\rangle \longrightarrow \langle c_1', \mu'\rangle}{\langle c_1; c_2, \mu\rangle \longrightarrow \langle c_1'; c_2, \mu'\rangle}\ \text{E-Seq}$$

$$\frac{\langle c_1, \mu\rangle \longrightarrow \textsf{failure}\ \mu'}{\langle c_1; c_2, \mu\rangle \longrightarrow \textsf{failure}\ \mu'}\ \text{E-SeqFail}$$

$$\frac{}{\langle \textbf{skip}; c_2, \mu\rangle \longrightarrow \langle c_2, \mu\rangle}\ \text{E-SeqSkip}$$

$$\frac{v \in \mathit{Val}}{\langle \textbf{havoc}\ x, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu[x := v]\rangle}\ \text{E-Havoc}$$

$$\frac{}{\langle x := e, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu[x := e(\mu)]\rangle}\ \text{E-Assign}$$

$$\frac{}{\langle c_1\ []\ c_2, \mu\rangle \longrightarrow \langle c_1, \mu\rangle}\ \text{E-ChoiceLeft}$$

$$\frac{}{\langle c_1\ []\ c_2, \mu\rangle \longrightarrow \langle c_2, \mu\rangle}\ \text{E-ChoiceRight}$$

Figure 4.2: Single Step Reduction Rules

Definition 4.2.5 (`VCG.step`). *The* single step reduction rules *are listed in Fig. 4.2. We say that σ' is a successor of σ if $\sigma \longrightarrow \sigma'$.*

We briefly clarify the meaning of the single step reduction rules:

- An **assert** e command is used to express target verification properties, meaning we expect e to evaluate to true on penalty of failure. The rules E-AssertTrue and E-AssertFalse reflect this: if e evaluates to true in the current store, the former applies and the assertion reduces to **skip**, allowing for further execution. If e evaluates to a different value, E-AssertFalse applies and execution ends up in a failure state.
- One of the uses of **assume** is to model language guarantees. For example, the this pointer is generally[2] guaranteed not to be null. Adding **assume** this $\neq$ null informs the verifier of this fact. Executing **assume** e entails checking if e evaluates to true in the current store. If it does, it amounts to a no-op (E-AssumeTrue). If e evaluates to false, execution gets stuck as there are no applicable reduction rules. Note that getting stuck does not equate with failure: it merely means this execution path need not be considered any further.
- Sequencing is dealt with in the expected way: the left command is reduced as much as possible (E-Seq). If it eventually reaches **skip**, it is dropped from the sequence (E-SeqSkip). If, however, it fails, so does the entire sequence (E-SeqFail).
- **havoc** x removes all information about x's binding (E-Havoc). This rule is nondeterministic, i.e. the rule associates more than one successor state to a given state. We further explain the meaning of this shortly.
- Assignment is conventional: it updates the store's binding (E-Assign).
- The choice command is another nondeterministic command: execution proceeds with either branch (E-ChoiceLeft, E-ChoiceRight). Choice is commonly used to model control flow, such as if statements or while loops: at verification time, it is generally impossible to determine which branch will be taken, so we ask the verifier to check both. Typically, **assume** commands are used to express in which conditions each path is taken, such as the knowledge that the then-clause of an if-statement is only executed in case the if-condition evaluated to true.

[1]The store component carries information that is used in some proofs.

[2]Notable exceptions to this are C++ and C♯'s extension methods.

```
translation(while $b$ invariant $I$ do $body$) =
    assert $I$;
    havoc syntactic targets of b
    assume $I$
    ((assume $b$; translation($body$); assert $I$; assume false) [] assume $\neg b$)
```

Figure 4.3: Translating a while loop

These commands should allow us to model all possible control flows. For example, while there is no direct support for modelling loops, we can encode them as shown in Fig. 4.3. First, loops need to be annotated with an invariant. The invariant must hold when execution reaches the loop: this is expressed by **assert** I. Next, we consider a "generalised loop iteration": since we don't know at verification time how many times a loop will be iterated over at runtime, we demand that a loop iteration preserves the invariant. That way, regardless of how many times the loop body is executed, we know that upon exiting it, the loop invariant will hold. The **havoc** destroys all information about the variables modified by the loop body. Information about their values is then restored using the loop invariant (**assume** I). Next, there are two choice branches. The left one models a generalised loop iteration: the loop condition can be assumed to hold (**assume** b), next the body is executed, and finally we check that the loop invariant still holds (**assert** I). Thus, in short, we can say that given $I \wedge b$ at the beginning of the l The **assume** false blocks further execution along this path. The right choice path models loop exit, hence we can assume that the loop condition evaluates to false (**assume** $\neg b$) on this execution path.

Definition 4.2.6 (`VCG.irreducible`). *We say a state σ is* irreducible *if it cannot be reduced any further.*

$$\text{irreducible}(\sigma) \quad \equiv \quad \neg(\exists\, \sigma.\ \sigma \longrightarrow \sigma')$$

As mentioned above, the reduction rules are nondeterministic: in some cases, more than one reduction rule applies, so that a state can have more than one successor. Nondeterminism gives rise to an *execution tree*: the initial execution state constitutes the root of the tree and a node's children are its successors. The leaves correspond with irreducible states. Every path from the root to a leaf represents an execution path.

We now define a multiple step relation which allow us to chain together multiple single steps.

Definition 4.2.7 (`VCG.steps`). *The* multiple step relation *groups zero or more steps together.*

$$\frac{}{\langle c, \mu\rangle \longrightarrow^* \langle c, \mu\rangle}\ \textsc{E*-Reflexive}$$

$$\frac{\langle c_1, \mu_1\rangle \longrightarrow \langle c_2, \mu_2\rangle \qquad \langle c_2, \mu_3\rangle \longrightarrow^* \langle c_3, \mu_3\rangle}{\langle c_1, \mu_1\rangle \longrightarrow^* \langle c_3, \mu_3\rangle}\ \textsc{E*-Step}$$

Detecting failures constitutes a major part of verification. Since we are dealing with nondeterministic execution, we need to take into account all possible execution paths, of which there are potentially infinitely many.

Definition 4.2.8 (`VCG.fails`). *We say a state σ* fails *if there exists an execution leading from σ to some failure state.*

$$\langle c, \mu\rangle \text{ fails} \quad \equiv \quad \exists\, \mu'.\, \sigma \longrightarrow^* \textsf{failure}\ \mu'$$

Definition 4.2.9 (`VCG.succeeds`). *A state σ* succeeds *if it does not fail, i.e. if there exists no path leading to failure.*

$$\langle c, \mu\rangle \text{ succeeds} \quad \equiv \quad \neg(\sigma \text{ fails})$$

In other words, a state σ succeeds iff the execution tree with root σ does not have a single failure leaf. We now prove a few lemmas about the reduction rules which will be needed in a later section.

Lemma 4.2.1 (`VCG.seq_skips`). *If a sequence $c_1; c_2$ reduces to* **skip***, we know that both c_1 and c_2 must reduce to* **skip** *separately. More formally, from*

$$\langle c_1; c_2, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle$$

follows that

$$\exists\, \mu''.\, \langle c_1, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu''\rangle \quad\wedge\quad \langle c_2, \mu''\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle$$

Proof. By induction on the derivation of $\langle c_1; c_2, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle$. The case for E*-Reflexive is trivially dealt with as $c_1; c_2$ cannot be equal to **skip**. Only E*-Step remains. We have the following proof state:

$$\frac{\begin{array}{ll} & \langle c_1; c_2, \mu\rangle \longrightarrow \sigma' \longrightarrow^* \langle \textbf{skip}, \mu'\rangle \\ H: & \forall\, c_{x,1}, c_{x,2}, \mu_x.\ \sigma' = \langle c_{x,1}; c_{x,2}, \mu_x\rangle \Rightarrow \\ & \qquad\qquad \exists\, \mu''.\ \ \langle c_{x,1}, \mu_x\rangle \longrightarrow^* \langle \textbf{skip}, \mu''\rangle \wedge \\ & \qquad\qquad\qquad\quad \langle c_{x,2}, \mu''\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle \end{array}}{\exists\, \mu''.\, \langle c_1, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu''\rangle \wedge \langle c_2, \mu''\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle}$$

$\langle c_1; c_2, \mu\rangle \longrightarrow \sigma'$ can only be the result of three reduction rules.

- E-Seq. We get $\sigma' = \langle c_1'; c_2, \mu'' \rangle$:

$$\frac{\begin{array}{rl} & \langle c_1, \mu \rangle \longrightarrow \langle c_1', \mu'' \rangle \\ & \langle c_1; c_2, \mu \rangle \longrightarrow \langle c_1'; c_2, \mu'' \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \\ H: & \forall\, c_{x,1}, c_{x,2}, \mu_x.\ \langle c_1'; c_2, \mu'' \rangle = \langle c_{x,1}; c_{x,2}, \mu_x \rangle \Rightarrow \\ & \qquad \exists\, \mu'''.\ \langle c_{x,1}, \mu_x \rangle \longrightarrow^* \langle \mathbf{skip}, \mu''' \rangle \wedge \\ & \qquad\qquad \langle c_{x,2}, \mu''' \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \end{array}}{\exists\, \mu''.\ \langle c_1, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle \wedge \langle c_2, \mu'' \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle}$$

  We specialise H with $c_{x,1} = c_1'$, $c_{x,2} = c_2$ and $\mu_x = \mu''$, which yields some store μ''' such that

$$\langle c_1, \mu \rangle \longrightarrow \langle c_1', \mu'' \rangle \longrightarrow^* \langle \mathbf{skip}, \mu''' \rangle \qquad \langle c_2, \mu''' \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle$$

  which is exactly what we need.

- E-SeqSkip: trivial, since $c_1 = \mathbf{skip}$ and $\sigma' = \langle c_2, \mu \rangle$.

$$\langle \mathbf{skip}; c_2, \mu \rangle \longrightarrow \langle c_2, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle$$

- E-SeqFail: since $\langle c_1, \mu \rangle \longrightarrow \textsf{failure}\ \mu''$ for some store μ'', we get as hypothesis

$$\langle c_1; c_2, \mu \rangle \longrightarrow \textsf{failure}\ \mu'' \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle$$

  This is impossible: failures are irreducible and hence can't reduce to **skip**.

□

Lemma 4.2.2 (`VCG.seq_fails`). *If a sequence $c_1; c_2$ reduces to failure, either c_1 or c_2 must fail. More formally, given*

$$\langle c_1; c_2, \mu \rangle \longrightarrow^* \textsf{failure}\ \mu'$$

then we know that

$$\langle c_1, \mu \rangle \longrightarrow^* \textsf{failure}\ \mu' \quad \vee \quad \exists\, \mu''.\ \langle c_1, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle \wedge \langle c_2, \mu'' \rangle \longrightarrow^* \textsf{failure}\ \mu'$$

Proof. By induction on the derivation of $\langle c_1; c_2, \mu \rangle \longrightarrow^* \textsf{failure}\ \mu'$. □

Lemma 4.2.3 (`VCG.append_steps`). *Transitivity of $\longrightarrow^*$: it is possible to chain steps together.*

$$\sigma \longrightarrow^* \sigma' \longrightarrow^* \sigma'' \quad \Rightarrow \quad \sigma \longrightarrow^* \sigma''$$

Proof. By induction on the derivation of $\sigma \longrightarrow^* \sigma'$. □

Lemma 4.2.4 (`VCG.steps_seq`). *We can lift* E-SEQ *to the multiple step relation.*

$$\langle c_1, \mu \rangle \longrightarrow^* \langle c_1', \mu' \rangle \quad \Rightarrow \quad \langle c_1; c_2, \mu \rangle \longrightarrow^* \langle c_1'; c_2, \mu' \rangle$$

Proof. By induction on the derivation of $\langle c_1, \mu \rangle \longrightarrow^* \langle c_1', \mu' \rangle$. □

Lemma 4.2.5 (`VCG.steps_seq_skip`).

$$\langle c_1, \mu \rangle \longrightarrow^* \langle \textbf{skip}, \mu' \rangle \quad \Rightarrow \quad \langle c_1; c_2, \mu \rangle \longrightarrow^* \langle c_2, \mu' \rangle$$

Proof. Follows from Lemma 4.2.4 and Lemma 4.2.3. □

4.3 Conclusion

In this chapter, we have formally defined the intermediate verification language which will be the subject of verification condition generation algorithms in the following chapters. The language cannot be used to perform computations: it is only meant to be used to model control flows and specify which target verification properties we wish to enforce along which path using the **assert** command.

Execution of programs written in the intermediate verification language can fail; this is due to an assertion condition not evaluating to true. The aim of verification is to prevent this from happening. For example, one way to accomplish this is to compute the set of initial program states for which execution will not fail. "Holes" in this set then indicate which inputs are not dealt with correctly. Full details will be given in the following chapter.

Chapter 5

Phase II: Generating Verification Conditions

Having formally defined our intermediate verification language (Chapter 4), we are able to express formally what program properties we wish to verify and how to generate verification conditions for them.

In this chapter, we examine multiple verification condition generation algorithms. First, Sect. 5.1 gives a short introduction to Hoare logic. Next, we discuss strongest postconditions (Sect. 5.2), weakest liberal preconditions (Sect. 5.3) and weakest preconditions (Sect. 5.4). For each algorithm, we prove its soundness, and where possible, relate them to each other. All definitions and theorems have been formalised in Coq (see Appendix D) and proofs are machine checked.

5.1 Hoare Logic

In Sect. 4.2, we have formally defined the behaviour of code written in our intermediate verification language using operational semantics. An alternative way to formalise semantics consists of specifying axiomatic semantics, Hoare Logic [57] being the canonical example.

Hoare logic's main feature is the *Hoare triple*, written $\{P\}\ c\ \{Q\}$, where c is a command and P and Q are (in our formalisation) store predicates named the precondition and postcondition, respectively. It represents the fact that if c is executed in a store satisfying P, it will transform the state into one satisfying Q.

Definition 5.1.1 (`VCG.store_predicate`). *A* store predicate *is the characteristic function of a set of stores.*

$$\text{store-predicate} \;\equiv\; \mathit{store} \rightarrow \text{Prop}$$

Prop is the type of propositions and for the purposes of this chapter can interpreted as equivalent to bool. A more elaborate explanation can be found in Sect. C.4.

Example 5.1.1. *Given a square root function $r = \text{sqrt}(x)$, its postcondition $r^2 = x$ would be modelled by the function $\lambda\ \mu.\ \mu(r)^2 = \mu(x)$, which represents the set of all stores where the given postcondition holds.*

Definition 5.1.2 (Hoare triple). *$\{P\}\ c\ \{Q\}$ expresses that executing c starting in a store satisfying P will not fail and reach an end store satisfying Q.*

$$\{P\}\ c\ \{Q\} \;\equiv\; \forall\ \mu, \mu'.\ P(\mu) \Rightarrow \langle c, \mu\rangle \text{ succeeds} \wedge \Big(\langle c, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle \Rightarrow Q(\mu')\Big)$$

Definition 5.1.3 (soft Hoare triple). *$|P|\ c\ |Q|$ expresses that all nonfailing executions starting in a store satisfying P end in a store satisfying Q.*

$$|P|\ c\ |Q| \quad\equiv\quad \forall\ \mu, \mu'.\ P(\mu) \Rightarrow \langle c, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle \Rightarrow Q(\mu')$$

$\{P\}\ c\ \{Q\}$ is stronger than $|P|\ c\ |Q|$ as it also disallows failure from happening. Note that the distinction between these two kinds of Hoare triples is unrelated to program termination, i.e. we are not dealing with partial versus total correctness. Programs written in the intermediate verification language (Def. 4.1.1, page 31) will always terminate, which means this distinction is irrelevant in our case.

Example 5.1.2. **skip** *is a no-op, meaning $\{P\}$* **skip** *$\{P\}$. Regarding sequencing of commands:*

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}}$$

Given a Hoare triple $\{P\}\ c\ \{Q\}$, we can wonder how we can adjust the precondition and postcondition so that it still remains a valid Hoare triple. We can always strengthen the precondition:

$$\frac{\{P'\}\ c\ \{Q\} \qquad P \Rightarrow P'}{\{P\}\ c\ \{Q\}}$$

However, this fails to yield interesting Hoare triples, as it throws information away. An extreme example would be to maximally strengthen the precondition, resulting in $\{\text{false}\}\ c\ \{Q\}$, which is trivially true for all c and Q.

Analogously, we can also weaken the postcondition:

$$\frac{\{P\}\ c\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}$$

which again amounts to discarding valuable information, e.g. in the extreme, it yields a rather meaningless $\{P\}\ c\ \{\text{true}\}$.

Doing the opposite, i.e. weakening the precondition and strengthening the postcondition, would be more useful, but one needs to be careful that the resulting Hoare triples are still valid. Finding the weakest precondition and the strongest postcondition is the subject of the following sections.

5.2 Strongest Postcondition

Finding the strongest postcondition $\text{sp}(c, P)$ for a given precondition P and command c consists of finding that postcondition for which

$$\forall\ Q'.\ |P|\ c\ |Q'| \Rightarrow \text{sp}(c, P) \Rightarrow Q' \qquad \wedge \qquad |P|\ c\ |\text{sp}(c, P)|$$

In terms of sets of states[1], this is equivalent with

$$\text{sp}(c, P) = \{\ \mu' \mid P(\mu) \wedge \langle c, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle\ \}$$

i.e. the strongest postcondition $\text{sp}(c, P)$ collects all states one can reach when executing c in a state described by P. It is the *strongest* postcondition because $\text{sp}(c, P)$ *only* contains those states.

[1]Since a term predicate is a set's characteristic function, we sometimes use a more intuitive set comprehension notation. In this example, it is possible to reformulate it as the following characteristic function:

$$\{\ \mu' \mid P(\mu) \wedge \langle c, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle\ \} \text{ denotes } \lambda\ \mu'.\ \exists\ \mu.\ P(\mu) \wedge \langle c, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle$$

Definition 5.2.1 (`VCG.strongest_postcondition`). *The strongest postcondition generation algorithm is defined as follows:*

$$
\begin{array}{rcl}
\mathrm{sp} & : & \text{command} \times \text{store-predicate} \rightarrow \text{store-predicate} \\
\hline
\mathrm{sp}(\textbf{assert } e, P) & = & \lambda\ \mu.\ e(\mu) \wedge P(\mu) \\
\mathrm{sp}(\textbf{assume } e, P) & = & \lambda\ \mu.\ e(\mu) \wedge P(\mu) \\
\mathrm{sp}(x := e, P) & = & \lambda\ \mu.\ \exists\ v.\ P(\mu[x := v]) \wedge \mu(x) = e(\mu[x := v]) \\
\mathrm{sp}(\textbf{havoc } x, P) & = & \lambda\ \mu.\ \exists\ v.\ P(\mu[x := v]) \\
\mathrm{sp}(\textbf{skip}, P) & = & P \\
\mathrm{sp}(c_1 \ [] \ c_2, P) & = & \lambda\ \mu.\ \mathrm{sp}(c_1, P)(\mu) \vee \mathrm{sp}(c_2, P)(\mu) \\
\mathrm{sp}(c_1; c_2, P) & = & \mathrm{sp}(c_2, \mathrm{sp}(c_1, P))
\end{array}
$$

The attentive reader might wonder why **assume** and **assert** produce the same verification condition. While we will of course formally prove this is indeed correct, an intuitive explanation goes as follows: the goal of the strongest precondition is to describe the set of all "output states" (i.e. states reaching **skip**) and given that both "failing" **assume**s and **assert**s keep execution from proceeding, the strongest postcondition must take care of filtering out those states that lead to the condition of the **assume** or **assert** command to evaluate to false. If a distinction between a failing **assume** and **assert** is to be made, the strongest postcondition algorithm needs to be adapted so that it returns more than just a selection of succeeding states, e.g. we could make it return two store predicates: one describing the set of all output states and a second describing all input states that lead to failure.

Example 5.2.1. *Consider*

$$c = \textbf{assume}\ \mathrm{x} \geq 0; \mathrm{y} := \mathrm{x} \quad [] \quad \textbf{assume}\ \mathrm{x} < 0; \mathrm{y} := -\mathrm{x}$$

We compute the strongest postcondition in steps (we use a more lightweight notation since this is just an example):

$$
\begin{array}{rcl}
\mathrm{sp}(\textbf{assume}\ \mathrm{x} \geq 0, P) & = & \mathrm{x} \geq 0 \wedge P \\
\mathrm{sp}(\mathrm{y} := \mathrm{x}, P) & = & \exists\ \mathrm{v}.\ P[\mathrm{v}/\mathrm{y}] \wedge \mathrm{y} = \mathrm{x} \\
\mathrm{sp}(\textbf{assume}\ \mathrm{x} \geq 0; \mathrm{y} := \mathrm{x}, P) & = & \exists\ \mathrm{v}.\ \mathrm{x} \geq 0 \wedge P[\mathrm{v}/\mathrm{y}] \wedge \mathrm{y} = \mathrm{x} \\
\mathrm{sp}(\textbf{assume}\ \mathrm{x} < 0, P) & = & \mathrm{x} < 0 \wedge P \\
\mathrm{sp}(\mathrm{y} := -\mathrm{x}, P) & = & \exists\ \mathrm{v}.\ P[\mathrm{v}/\mathrm{y}] \wedge \mathrm{y} = -\mathrm{x} \\
\mathrm{sp}(\mathrm{x} < 0; \mathrm{y} := -\mathrm{x}, P) & = & \exists\ \mathrm{v}.\ \mathrm{x} < 0 \wedge P[\mathrm{v}/\mathrm{y}] \wedge \mathrm{y} = -\mathrm{x} \\
\mathrm{sp}(c, P) & = & \exists\ \mathrm{v}.\ \ P[\mathrm{v}/\mathrm{y}] \wedge \\
 & & \qquad (\mathrm{x} \geq 0 \wedge \mathrm{y} = \mathrm{x}) \vee \\
 & & \qquad (\mathrm{x} < 0 \wedge \mathrm{y} = -\mathrm{x}) \\
 & = & \exists\ \mathrm{v}.\ P[\mathrm{v}/\mathrm{y}] \wedge \mathrm{y} = |\mathrm{x}|
\end{array}
$$

Thus, if we consider an arbitrary state (represented by taking true *as precondition, which allows any state), we get*

$$\mathrm{sp}(c, \mathrm{true}) \; = \; \mathrm{y} = |\mathrm{x}|$$

We will now prove that the strongest postcondition generation works as advertised. On the one hand, we wish them to yield postconditions; on the other hand, we wish them to be the strongest.

Theorem 5.2.1 (`VCG.sp_soundness`)**.** *The strongest postcondition algorithm is sound. If execution of a command c in a store satisfying a store predicate P ends up in a state* $\langle \mathbf{skip}, \mu' \rangle$*, then* μ' *satisfies* $\mathrm{sp}(c, P)(\mu')$*.*

$$P(\mu) \Rightarrow \langle c, \mu \rangle \longrightarrow^{*} \langle \mathbf{skip}, \sigma' \rangle \Rightarrow \mathrm{sp}(c, P)(\mu')$$

Proof. By structural induction on c and Lemma 4.2.1 for dealing with sequencing. □

Lemma 5.2.1 (`VCG.sp_strongest_aux`)**.** *A state* μ' *allowed by the strongest postcondition* $\mathrm{sp}(c, P)$ *is reachable from a state* $\langle c, \mu \rangle$ *where* μ *satisfies the precondition.*

$$\mathrm{sp}(c, P)(\mu') \Rightarrow \exists\ \mu.\ P(\mu) \wedge \langle c, \mu \rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu' \rangle$$

Proof. By structural induction on c, relying on Lemma 4.2.5 and Lemma 4.2.3 for the sequence case. □

Theorem 5.2.2 (`VCG.sp_strongest`)**.** *Given a command c, a precondition P and a postcondition Q such that*

$$\forall\ \mu, \mu'.\ P(\mu) \Rightarrow \langle c, \mu \rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu' \rangle \Rightarrow Q(\mu')$$

i.e. Q "catches" all states produced by executing c starting off in P. Then

$$\forall\ \mu'.\ \mathrm{sp}(c, P)(\mu') \Rightarrow Q(\mu')$$

i.e. the strongest postcondition is stronger than Q.

Proof. We start with the following proof state:

$$\begin{array}{ll} H: & \forall\ \mu_{\mathrm{x}}, \mu'_{\mathrm{x}}.\ P(\mu_{\mathrm{x}}) \Rightarrow \langle c, \mu_{\mathrm{x}} \rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu'_{\mathrm{x}} \rangle \Rightarrow Q(\mu'_{\mathrm{x}}) \\ & \mathrm{sp}(c, P)(\mu') \\ \hline & Q(\mu') \end{array}$$

Using Lemma 5.2.1, we know of some store μ such that

$$\begin{array}{ll} H: & \forall\, \mu_{\mathrm{x}}, \mu'_{\mathrm{x}}.\ P(\mu_{\mathrm{x}}) \Rightarrow \langle c, \mu_{\mathrm{x}}\rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu'_{\mathrm{x}}\rangle \Rightarrow Q(\mu'_{\mathrm{x}}) \\ & \mathrm{sp}(c, P)(\mu') \\ & P(\mu) \\ & \langle c, \mu\rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu'\rangle \\ \hline & Q(\mu') \end{array}$$

We apply H with $\mu_{\mathrm{x}} = \mu$ and $\mu'_{\mathrm{x}} = \mu'$. □

The strongest postcondition allows us to verify the output of successful executions, i.e. executions ending in some state $\langle \mathbf{skip}, \mu'\rangle$ for some store μ'. However, they do not give us a means to detect failing execution paths.

Example 5.2.2. *We consider the command*

$$c = \mathbf{assert}\ \mathrm{x} \geq 0\ []\ \mathbf{assert}\ \mathrm{x} \leq 0$$

The only path which does not fail is the one that starts with a store which binds x to 0*. Let us examine if this is reflected in the strongest postcondition. The strongest postcondition is*

$$\begin{array}{rcl} \mathrm{sp}(c, P) & \Longleftrightarrow & (\mathrm{x} \geq 0 \vee \mathrm{x} \leq 0) \wedge P \\ & \Longleftrightarrow & P \\ & \Longleftrightarrow & \mathrm{sp}(\mathbf{skip}, P) \end{array}$$

Thus, strongest postconditions can make no distinction between the potentially failing **assert** $x \geq 0$ [] **assert** $x \leq 0$ *and the always succeeding* **skip**.

5.3 Weakest Liberal Precondition

The strongest postconditions we discussed in Sect. 5.2 produced a postcondition based on a precondition, i.e. reasoning proceeded in the forward direction. In this section, we turn this around: given a postcondition, the weakest liberal postcondition wlp(c, Q) is that precondition for which

$$\forall\, P'.\ |P'|\ c\ |Q| \Rightarrow P' \Rightarrow \mathrm{wlp}(c, Q) \qquad \wedge \qquad |\mathrm{wlp}(c, Q)|\ c\ |Q|$$

Definition 5.3.1 (`VCG.weakest_liberal_precondition`). *The* weakest liberal precondition *algorithm is defined as follows:*

$$\begin{array}{rcl}
\text{wlp} & : & \text{command} \times \text{store-predicate} \to \text{store-predicate} \\
\hline
\text{wlp}(\textbf{assert}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \Rightarrow Q(\mu) \\
\text{wlp}(\textbf{assume}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \Rightarrow Q(\mu) \\
\text{wlp}(x := e, Q) & = & \lambda\ \mu.\ Q(\mu[x := e(\mu)]) \\
\text{wlp}(\textbf{havoc}\ x, Q) & = & \lambda\ \mu.\ \forall\ v.\ Q(\mu[x := v]) \\
\text{wlp}(\textbf{skip}, Q) & = & Q \\
\text{wlp}(c_1\ []\ c_2, Q) & = & \lambda\ \mu.\ \text{wlp}(c_1, Q)(\mu) \wedge \text{wlp}(c_2, Q)(\mu) \\
\text{wlp}(c_1; c_2, Q) & = & \text{wlp}(c_1, \text{wlp}(c_2, Q))
\end{array}$$

Example 5.3.1. *Let us take the same program as in Example 5.2.1:*

$$c = \textbf{assume}\ \mathrm{x} \geq 0; \mathrm{y} := \mathrm{x} \quad [] \quad \textbf{assume}\ \mathrm{x} < 0; \mathrm{y} := -\mathrm{x}$$

The weakest liberal precondition is

$$\begin{array}{rcl}
\text{wlp}(\textbf{assume}\ \mathrm{x} \geq 0, Q) & = & \mathrm{x} \geq 0 \Rightarrow Q \\
\text{wlp}(\mathrm{y} := \mathrm{x}, Q) & = & Q[\mathrm{x}/\mathrm{y}] \\
\text{wlp}(\textbf{assume}\ \mathrm{x} \geq 0; \mathrm{y} := \mathrm{x}, Q) & = & \mathrm{x} \geq 0 \Rightarrow Q[\mathrm{x}/\mathrm{y}] \\
\text{wlp}(\textbf{assume}\ \mathrm{x} < 0, Q) & = & \mathrm{x} < 0 \Rightarrow Q \\
\text{wlp}(\mathrm{y} := -\mathrm{x}, Q) & = & Q[-\mathrm{x}/\mathrm{y}] \\
\text{wlp}(\textbf{assume}\ \mathrm{x} < 0; \mathrm{y} := -\mathrm{x}, Q) & = & \mathrm{x} < 0 \Rightarrow Q[-\mathrm{x}/\mathrm{y}] \\
\text{wlp}(c, Q) & = & (\mathrm{x} \geq 0 \Rightarrow Q[\mathrm{x}/\mathrm{y}]) \wedge (\mathrm{x} < 0 \Rightarrow Q[-\mathrm{x}/\mathrm{y}]) \\
 & = & Q[|\mathrm{x}|/\mathrm{y}]
\end{array}$$

If our expectation of c is to produce a positive y, we get

$$\text{wlp}(c, y \geq 0) \ = \ |x| \geq 0$$

which is a tautology, i.e. no matter what input we feed our program, y will come out positive. If we strengthen our expectations, namely that we wish y to be equal to the absolute value of x, we get

$$\text{wlp}(c, y = |x|) \ = \ |x| = |x|$$

which again is a tautology. If we make any stronger demands however,

$$\text{wlp}(c, y = 3) \ = \ |x| = 3$$

Thus, we can relate this to the strongest postcondition by noting that deriving the wlp *amounts to finding the strongest Q for which* wlp(c, Q) *is valid.*

Theorem 5.3.1 (`VCG.wlp_soundness`). *The weakest liberal precondition algorithm is sound. Given a command c and a postcondition Q, if we start execution of c in a state satisfying* wlp(c, Q)*, we will end up in a state satisfying Q.*

$$\text{wlp}(c, Q)(\mu) \quad \Rightarrow \quad \langle c, \mu \rangle \longrightarrow^* \langle \textbf{skip}, \mu' \rangle \quad \Rightarrow \quad Q(\mu')$$

Proof. By structural induction on c and Lemma 4.2.1 for sequences. □

Note this theorem focuses solely on executions which end in **skip**, i.e. failures are ignored. The weakest liberal preconditions suffer the same problem as the strongest preconditions: they don't allow us to prevent failures.

Example 5.3.2. *We examine the same command as in Example 5.2.2.*

$$c = \textbf{assert}\ x \geq 0\ []\ \textbf{assert}\ x \leq 0$$

The weakest liberal precondition is

$$\begin{array}{rcl} \text{wlp}(c, Q) & = & (x \geq 0 \Rightarrow Q) \wedge (x \leq 0 \Rightarrow Q) \\ & = & Q \\ & = & \text{wlp}(\textbf{skip}, Q) \end{array}$$

As with the strongest postcondition, the weakest liberal postcondition does not distinguish **assert** $x \geq 0$ [] **assert** $x \leq 0$ *from* **skip**.

We now relate the weakest liberal precondition with the strongest postcondition, from which follows that wlp does indeed yield the weakest possible predicate.

Lemma 5.3.1 (`VCG.wlp_sp`)**.** wlp *and* sp *are related as follows:*

$$(\forall\ \mu.\ \text{sp}(c, P)(\mu) \Rightarrow Q(\mu)) \quad \Rightarrow \quad P(\mu) \Rightarrow \text{wlp}(c, Q)(\mu)$$

Proof. By structural induction on c. □

Theorem 5.3.2 (`VCG.wlp_weakest`)**.** *Given a precondition P and a postcondition Q, if execution starting in any state satisfying P ends up in a state satisfying Q:*

$$\forall\ \mu, \mu'.\ P(\mu) \Rightarrow \langle c, \mu \rangle \longrightarrow^* \langle \textbf{skip}, \mu' \rangle \Rightarrow Q(\mu')$$

then P is stronger than the weakest liberal precondition:

$$\forall\ \mu.\ P(\mu) \Rightarrow \text{wlp}(c, Q)(\mu)$$

Proof. From Theorem 5.2.2, we know that from

$$\forall\ \mu, \mu'.\ P(\mu) \Rightarrow \langle c, \mu \rangle \longrightarrow^* \langle \textbf{skip}, \mu' \rangle \Rightarrow Q(\mu')$$

follows that

$$\forall\ \mu'.\ \text{sp}(c, P)(\mu') \Rightarrow Q(\mu')$$

This combined with Lemma 5.3.1 finishes the proof. □

5.4 Weakest Precondition

The weakest liberal precondition, as discussed in 5.3, cannot be used to detect failures. In this section, we define the weakest precondition algorithm which provide a remedy for this shortcoming.

The weakest precondition $\mathrm{wp}(c, Q)$ makes an extra guarantee compared to the weakest liberal precondition: for all executions starting in a wp-satisfying initial store, not only will all successful paths (i.e. ending in **skip**) end up with a store satisfying Q, it is also guaranteed that no execution path will encounter failure. Using Hoare triples, we can write

$$\forall\, P.\ \{P\}\ c\ \{Q\} \Rightarrow P \Rightarrow \mathrm{wp}(c, Q) \qquad \wedge \qquad \{\mathrm{wp}(c, Q)\}\ c\ \{Q\}$$

Definition 5.4.1 (`VCG.weakest_precondition`)**.** *The* weakest precondition *algorithm is defined as follows:*

$$\begin{array}{rcl}
\mathrm{wp} & : & \text{command} \times \text{store-predicate} \to \text{store-predicate} \\
\hline
\mathrm{wp}(\textbf{assert}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \wedge Q(\mu) \\
\mathrm{wp}(\textbf{assume}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \Rightarrow Q(\mu) \\
\mathrm{wp}(x := e, Q) & = & \lambda\ \mu.\ Q(\mu[x := e(\mu)]) \\
\mathrm{wp}(\textbf{havoc}\ x, Q) & = & \lambda\ \mu.\ \forall\ v.\ Q(\mu[x := v]) \\
\mathrm{wp}(\textbf{skip}, Q) & = & Q \\
\mathrm{wp}(c_1\ []\ c_2, Q) & = & \lambda\ \mu.\ \mathrm{wp}(c_1, Q)(\mu) \wedge \mathrm{wp}(c_2, Q)(\mu) \\
\mathrm{wp}(c_1; c_2, Q) & = & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))
\end{array}$$

Example 5.4.1. *We examine the same command as in Example 5.2.2 and Example 5.3.2:*

$$c = \textbf{assert}\ \mathrm{x} \geq 0\ []\ \textbf{assert}\ \mathrm{x} \leq 0$$

The weakest precondition is

$$\begin{array}{rcl}
\mathrm{wp}(c, Q) & = & (\mathrm{x} \geq 0 \wedge Q) \wedge (\mathrm{x} \leq 0 \wedge Q) \\
& = & \mathrm{x} = 0 \wedge Q
\end{array}$$

For example, if we make no demands about the postcondition,

$$\mathrm{wp}(c, \mathrm{true}) \ = \ \mathrm{x} = 0$$

This clearly reflects that successful execution depends on $\mathrm{x} = 0$*. We will come back to this later.*

Definition 5.4.2 (`VCG.final_state`)**.** *We say a state σ is* final *if it is either a failure state or an in-progress state with* **skip** *as command component.*

$$\begin{array}{rcl}
\mathsf{final}(\mathsf{failure}\ \mu) & = & \mathrm{true} \\
\mathsf{final}(\langle c, \mu \rangle) & = & c = \textbf{skip}
\end{array}$$

Lemma 5.4.1 (`VCG.irreducible_final_state`)**.** *Final states are irreducible.*

$$\forall\ \sigma.\ \text{final}(\sigma) \quad \Rightarrow \quad \text{irreducible}(\sigma)$$

Definition 5.4.3 (`VCG.bigstep`)**.** *The* big step relation *relates a state to reachable final states.*

$$\sigma \downarrow_{\mathrm{f}} \sigma' \quad \equiv \quad \sigma \longrightarrow^{*} \sigma' \ \wedge\ \text{final}(\sigma')$$

Lemma 5.4.2 (`VCG.wp_skip`)**.** *Given a state $\sigma = \langle c, \mu\rangle$ and a store predicate Q. If the initial store μ satisfies the weakest precondition* $\text{wp}(c, Q)$*, the big step relation will only lead to non-failure states whose store satisfies Q. More formally,*

$$\forall\ c, Q, \mu, \sigma.\ \text{wp}(c, Q)(\mu) \ \Rightarrow\ \langle c, \mu\rangle \downarrow_{\mathrm{f}} \sigma \ \Rightarrow\ \exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu')$$

Proof. By structural induction on c:

- $c = \textbf{skip}$: trivial.
- $c = \textbf{assert}\ e$. The proof state becomes

$$\begin{array}{l} e(\mu) \wedge Q(\mu) \\ \langle \textbf{assert}\ e, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ \hline \exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu') \end{array}$$

  There are two applicable reduction routes: through E-AssertTrue and E-AssertFalse (see Fig. 4.2).

  - E-AssertTrue.

$$\begin{array}{l} e(\mu) \wedge Q(\mu) \\ \langle \textbf{assert}\ e, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ \sigma = \langle \textbf{skip}, \mu\rangle \\ \hline \exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu') \end{array}$$

    We pick $\mu' = \mu$.

  - E-AssertFalse. This rule only applies when $\neg e(\mu)$:

$$\begin{array}{l} e(\mu) \wedge Q(\mu) \\ \neg e(\mu) \\ \langle \textbf{assert}\ e, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ \sigma = \mathsf{failure}\ \mu \\ \hline \exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu') \end{array}$$

    There is a clear contradiction in the hypotheses, allowing us to ignore this case.

- c = **assume** e.

$$\frac{\begin{array}{l} e(\mu) \Rightarrow Q(\mu) \\ \langle \textbf{assume}\ e, \mu\rangle \downarrow_{\mathrm{f}} \sigma \end{array}}{\exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

  There is only one possible path, namely through E-AssumeTrue. This rule only applies when $e(\mu)$ yields true.

$$\frac{\begin{array}{l} e(\mu) \Rightarrow Q(\mu) \\ \langle \textbf{assume}\ e, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ e(\mu) \\ \sigma = \langle \textbf{skip}, \mu\rangle \end{array}}{\exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

  We pick $\mu' = \mu$.

- c = **havoc** x. Only E-Havoc applies:

$$\frac{\begin{array}{l} \forall\ v.\ Q(\mu[x := v]) \\ \langle \textbf{havoc}\ x, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ v \in \mathrm{val} \\ \sigma = \langle \textbf{skip}, \mu[x := v]\rangle \end{array}}{\exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

  We pick $\mu' = \mu[x := v]$.

- $c = x := e$. Only E-Assign applies:

$$\frac{\begin{array}{l} Q(\mu[x := e(\mu)]) \\ \langle x{:=}e, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ \sigma = \langle \textbf{skip}, \mu[x := e(\mu)]\rangle \end{array}}{\exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

  We pick $\mu' = \mu[x := e(\mu)]$.

- $c = c_1\ []\ c_2$. Induction yields two induction hypotheses H_1 and H_2:

$$\frac{\begin{array}{rl} & \mathrm{wp}(c_1, Q)(\mu) \wedge \mathrm{wp}(c_2, Q)(\mu) \\ & \langle c_1\ []\ c_2, \mu\rangle \downarrow_{\mathrm{f}} \sigma \\ H_1: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \textbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\ H_2: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \textbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \end{array}}{\exists\ \mu'.\ \sigma = \langle \textbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

We can distinguish two reduction paths: one through E-ChoiceLeft, the other through E-ChoiceRight. We consider only one, the second can be dealt with analogously.

$$\begin{array}{rl}
& \mathrm{wp}(c_1, Q)(\mu) \wedge \mathrm{wp}(c_2, Q)(\mu) \\
& \langle c_1 \; [] \; c_2, \mu \rangle \longrightarrow \langle c_1, \mu \rangle \\
& \langle c_1, \mu \rangle \downarrow_{\mathrm{f}} \sigma \\
H_1 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}} \rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
& \qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu' \rangle \wedge Q_{\mathrm{x}}(\mu') \\
H_2 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}} \rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
& \qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu' \rangle \wedge Q_{\mathrm{x}}(\mu') \\
\hline
& \exists\, \mu'.\ \sigma = \langle \mathbf{skip}, \mu' \rangle \wedge Q(\mu')
\end{array}$$

The goal follows from the induction hypothesis H_1, taking $Q_{\mathrm{x}} = Q$, $\mu_{\mathrm{x}} = \mu$ and $\sigma_{\mathrm{x}} = \sigma$.

- $c = c_1; c_2$.

$$\begin{array}{rl}
& \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\
H_0 : & \langle c_1; c_2, \mu \rangle \downarrow_{\mathrm{f}} \sigma \\
H_1 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}} \rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
& \qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu' \rangle \wedge Q_{\mathrm{x}}(\mu') \\
H_2 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}} \rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
& \qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu' \rangle \wedge Q_{\mathrm{x}}(\mu') \\
\hline
& \exists\, \mu'.\ \sigma = \langle \mathbf{skip}, \mu' \rangle \wedge Q(\mu')
\end{array}$$

We know σ to be a final state; we distinguish two cases:

  - $\sigma = \langle \mathbf{skip}, \mu_{\mathrm{f}} \rangle$ for some store μ_{f}. In the goal, we pick $\mu' = \mu_{\mathrm{f}}$.

$$\begin{array}{rl}
& \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\
H_0 : & \langle c_1; c_2, \mu \rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu_{\mathrm{f}} \rangle \\
H_1 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}} \rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
& \qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu' \rangle \wedge Q_{\mathrm{x}}(\mu') \\
H_2 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}} \rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
& \qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu' \rangle \wedge Q_{\mathrm{x}}(\mu') \\
\hline
& \langle \mathbf{skip}, \mu_{\mathrm{f}} \rangle = \langle \mathbf{skip}, \mu_{\mathrm{f}} \rangle \wedge Q(\mu_{\mathrm{f}})
\end{array}$$

The goal's left conjunct is trivial. With the help of Lemma 4.2.1 we can split H_0 into two parts:

$$\begin{array}{rl}
 & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\
 & \langle c_1, \mu\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu''\rangle \\
 & \langle c_2, \mu''\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu_{\mathrm{f}}\rangle \\
H_1 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
 & \qquad\qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\
H_2 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
 & \qquad\qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\
\hline
 & Q(\mu_{\mathrm{f}})
\end{array}$$

Specialising H_1 with $Q_{\mathrm{x}} = \mathrm{wp}(c_2, Q)$, $\mu_{\mathrm{x}} = \mu$ and $\sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu''\rangle$ yields

$$\begin{array}{rl}
 & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\
 & \langle c_1, \mu\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu''\rangle \\
 & \langle c_2, \mu''\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu_{\mathrm{f}}\rangle \\
H_1 : & \mathrm{wp}(c_2, Q)(\mu'') \\
H_2 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
 & \qquad\qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\
\hline
 & Q(\mu_{\mathrm{f}})
\end{array}$$

We can now specialise H_2 with $Q_{\mathrm{x}} = Q$, $\mu_{\mathrm{x}} = \mu''$ and $\sigma_{\mathrm{x}} = \langle \mathbf{skip}, \sigma_{\mathrm{f}}\rangle$.

$$\begin{array}{rl}
 & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\
 & \langle c_1, \mu\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu''\rangle \\
 & \langle c_2, \mu''\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu_{\mathrm{f}}\rangle \\
H_1 : & \mathrm{wp}(c_2, Q)(\mu'') \\
H_2 : & Q(\mu_{\mathrm{f}}) \\
\hline
 & Q(\mu_{\mathrm{f}})
\end{array}$$

– $\sigma = \mathsf{failure}\ \mu_{\mathrm{f}}$ for some store μ_{f}. We show this leads to a contradiction.

$$\begin{array}{rl}
 & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\
H_0 : & \langle c_1; c_2, \mu\rangle \downarrow_{\mathrm{f}} \mathsf{failure}\ \mu_{\mathrm{f}} \\
H_1 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
 & \qquad\qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\
H_2 : & \forall\, Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\
 & \qquad\qquad \exists\, \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\
\hline
 & \exists\, \mu'.\ \sigma = \langle \mathbf{skip}, \mu'\rangle \wedge Q(\mu')
\end{array}$$

Using Lemma 4.2.2, we again distinguish two cases: either c_1 fails, or c_1 reduces to skip and c_2 fails.

* c_1 fails.

$$\frac{\begin{array}{rl} & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\ & \langle c_1, \mu\rangle \downarrow_{\mathrm{f}} \textsf{failure}\ \mu_{\mathrm{f}} \\ H_1: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\ H_2: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \end{array}}{\exists\ \mu'.\ \sigma = \langle \mathbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

Specialising H_1 with $Q_{\mathrm{x}} = \mathrm{wp}(c_2, Q)$, $\mu_{\mathrm{x}} = \mu$ and $\sigma_{\mathrm{x}} = \textsf{failure}\ \mu_{\mathrm{f}}$ leads to $\exists\ \mu'.\ \textsf{failure}\ \mu_{\mathrm{f}} = \langle \mathbf{skip}, \mu'\rangle$, which is absurd.

* c_1 skips, c_2 fails.

$$\frac{\begin{array}{rl} & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\ & \langle c_1, \mu\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu''\rangle \\ & \langle c_2, \mu''\rangle \downarrow_{\mathrm{f}} \textsf{failure}\ \mu_{\mathrm{f}} \\ H_1: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_1, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_1, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \\ H_2: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \end{array}}{\exists\ \mu'.\ \sigma = \langle \mathbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

We specialise H_1 with $Q_{\mathrm{x}} = \mathrm{wp}(c_2, Q)$, $\mu_{\mathrm{x}} = \mu$ and $\sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu''\rangle$.

$$\frac{\begin{array}{rl} & \mathrm{wp}(c_1, \mathrm{wp}(c_2, Q))(\mu) \\ & \langle c_1, \mu\rangle \downarrow_{\mathrm{f}} \langle \mathbf{skip}, \mu''\rangle \\ & \langle c_2, \mu''\rangle \downarrow_{\mathrm{f}} \textsf{failure}\ \mu_{\mathrm{f}} \\ H_1: & \mathrm{wp}(c_2, Q)(\mu'') \\ H_2: & \forall\ Q_{\mathrm{x}}, \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}.\ \mathrm{wp}(c_2, Q_{\mathrm{x}})(\mu_{\mathrm{x}}) \Rightarrow \langle c_2, \mu_{\mathrm{x}}\rangle \downarrow_{\mathrm{f}} \sigma_{\mathrm{x}} \Rightarrow \\ & \qquad \exists\ \mu'.\ \sigma_{\mathrm{x}} = \langle \mathbf{skip}, \mu'\rangle \wedge Q_{\mathrm{x}}(\mu') \end{array}}{\exists\ \mu'.\ \sigma = \langle \mathbf{skip}, \mu'\rangle \wedge Q(\mu')}$$

Specialising H_2 with $Q_{\mathrm{x}} = Q$, $\mu_{\mathrm{x}} = \mu''$ and $\sigma_{\mathrm{x}} = \textsf{failure}\ \mu_{\mathrm{f}}$ yields $\exists\ \mu'.\ \textsf{failure}\ \mu_{\mathrm{f}} = \langle \mathbf{skip}, \mu'\rangle$, which is absurd.

□

Theorem 5.4.1 (`VCG.wp_nofail`)**.** *Given a program c, its evaluation does not fail if the initial store μ satisfies the weakest precondition.*

$$\forall\ c, \mu.\ \mathrm{wp}(\mu, c)(\mathrm{true}) \quad \Rightarrow \quad \langle c, \mu\rangle\ \text{succeeds}$$

Proof. We must show that $\langle c, \mu\rangle$ never reduces to a failure state. Let us assume it does and show this leads to a contradiction: let's say there exists a store μ' such that $\langle c, \mu\rangle \longrightarrow^* \mathsf{failure}\ \mu'$. According to Def. 5.4.2, the right hand state constitutes a final state. Definition 5.4.3 then yields $\langle c, \mu\rangle \downarrow_f \mathsf{failure}\ \mu'$. From Lemma 5.4.2 follows that there exists some store μ'' such that $\mathsf{failure}\ \mu' = \langle \mathbf{skip}, \mu''\rangle$, which is absurd. □

Theorem 5.4.2 (`VCG.wp_postcondition`)**.** *Given a program c and a postcondition Q, any succeeding (i.e. ending in* **skip***) execution beginning with a store satisfying* $\mathrm{wp}(c, Q)$ *will end with a store satisfying Q.*

$$\mathrm{wp}(c, Q)(\mu) \quad \Rightarrow \quad \langle c, \mu\rangle \longrightarrow^* \langle \mathbf{skip}, \mu'\rangle \quad \Rightarrow \quad Q(\mu')$$

Proof. Follows from Lemma 5.4.2. □

Theorem 5.4.1 and Theorem 5.4.2 state two aspects of the weakest preconditions: they prevent failures (as already mentioned in Example 5.4.1) and derive that precondition such that execution ends satisfying a specified postcondition, respectively. We make this explicit in Theorem 5.4.3. This property and others will prove important in Chapter 6.

Lemma 5.4.3 (`VCG.wp_monotonic`)**.** *Monotonicity of the weakest preconditions: given postconditions Q and Q′ for which* $Q(\mu) \Rightarrow Q'(\mu)$ *for any store μ, then for any command c,*

$$\forall\ \mu.\ \mathrm{wp}(c, Q)(\mu) \Rightarrow \mathrm{wp}(c, Q')(\mu)$$

Proof. By structural induction on c. □

Lemma 5.4.4 (`VCG.wlp_monotonic`)**.** *Monotonicity of weakest liberal precondition: given postconditions Q and Q′ for which*

$$\forall\ \mu.\ Q(\mu) \Rightarrow Q'(\mu)$$

then for any command c,

$$\forall\ \mu.\ \mathrm{wlp}(c, Q)(\mu) \Rightarrow \mathrm{wlp}(c, Q')(\mu)$$

Proof. By structural induction on c. □

Lemma 5.4.5 (`VCG.wp_impl_wlp`)**.** *For all commands c and postconditions Q,*

$$\forall\ \mu.\ \mathrm{wp}(c, Q)(\mu) \Rightarrow \mathrm{wlp}(c, Q)(\mu)$$

Proof. By structural induction on c, the sequence case using Lemma 5.4.4. □

Lemma 5.4.6 (`VCG.wlp_conj`). *A conjunction of weakest liberal preconditions implies the weakest liberal precondition of the conjunction of their postconditions. For any command c and postconditions Q and Q′,*

$$\forall\ \mu.\ \mathrm{wlp}(c, Q)(\mu) \Rightarrow \mathrm{wlp}(c, Q')(\mu) \Rightarrow \mathrm{wlp}(c, Q \wedge Q')(\mu)$$

Proof. By structural induction on c. □

Theorem 5.4.3 (`VCG.wp_as_wlp_and_nf`). *The weakest precondition can be split into nonfailure and postcondition establishment.*

$$\forall\ \mu.\ \mathrm{wp}(c, Q)(\mu) \iff \mathrm{wlp}(c, Q)(\mu) \wedge \mathrm{wp}(c, \mathrm{true})(\mu)$$

Proof. The $\Rightarrow$ part follows directly from Lemma 5.4.3 and Lemma 5.4.5. The $\Leftarrow$ part is proven by structural induction on c with the help of Lemma 5.4.6. □

5.5 Conclusion

In this chapter, we have discussed three verification condition generation algorithms. The first, the strongest postcondition, reasons in the forward direction: given a set of initial states (the precondition), it produces all reachable states (the postcondition). However, it cannot discern successful execution from failing ones, making it unsuitable for the purposes of verification.

The second verification condition generation algorithm which we discussed was the weakest liberal preconditions. Contrary to the strongest postcondition, it works backwards: it produces the set of initial states (a precondition) for which execution will end in a state satisfying a given postcondition. It has the same shortcoming as the strongest postcondition, namely it does not detect failure.

Lastly, we discussed the weakest precondition algorithm, which operates similarly to the weakest liberal precondition but has the added ability to prevent failure, making it well suited for verification.

These algorithms are well established in the domain of software verification and many verifiers rely on them, which makes it all the more important to ascertain their soundness. Our contribution consists of having implemented these algorithms and having proved their soundness in Coq.

Chapter 6

Efficient Weakest Preconditions

In Chapter 5 we discussed several verification condition generation algorithms, of which the weakest precondition algorithm stood out thanks to its ability to prevent failures whereas the others could not. However, it still suffers from a major weakness: the generated verification condition grows exponentially with the size of the program. This is due to the way the choice command is handled:

$$\mathrm{wp}(c_1 \ [] \ c_2, Q) = \lambda\ \mu.\ \mathrm{wp}(c_1, Q)(\mu) \wedge \mathrm{wp}(c_2, Q)(\mu)$$

It effectively duplicates the postcondition Q, which is the cause of the exponential blowup.

In this section we develop an implementation of the "efficient weakest preconditions" described by Leino [77], which is essentially a reformulation of Flanagan and Saxe [52] and has the advantage of growing polynomially instead of exponentially. After full formalisation, we prove its soundness and show that its complexity is indeed polynomial.

However, the efficient weakest preconditions algorithm depends on the fact that programs are passive, or in other words, that they do not contain assignments. To circumvent this limitation, we define a way to transform an arbitrary program into an equivalent passified form. This is done in two phases:

1. The first phase transforms an arbitrary program into single assignment form and is the subject of Sect. 6.1.

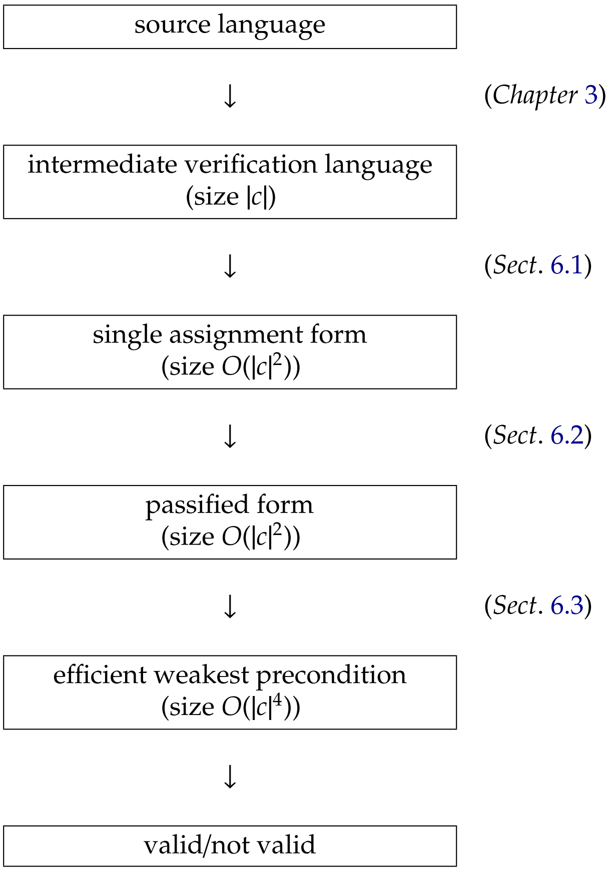
source language
(Chapter 3)
intermediate verification language (size $|c|$)
(Sect. 6.1)
single assignment form (size $O(|c|^2)$)
(Sect. 6.2)
passified form (size $O(|c|^2)$)
(Sect. 6.3)
efficient weakest precondition (size $O(|c|^4)$)
valid/not valid

Figure 6.1: Overview

2. The second phase performs the actual passification by removing all assignments. It is discussed in detail in Sect. 6.2.

In Sect. 6.3, we describe how to generate weakest preconditions efficiently from passified programs. Finally, in Sect. 6.4, we prove the soundness of this approach, i.e. we show that each transformation produces an equivalent program for the purposes of verification, and thus that the generated verification condition is meaningful for the original program. Figure 6.1 gives a complete overview of all the steps needed to verify a program using efficient weakest preconditions.

A note regarding **havoc**: our original work [112] did not consider this command

and hence no Coq proofs which take **havoc** into consideration are available. Due to time constraints we have not been able to update the Coq script. For this chapter, we have chosen to remain as consistent as possible with the Coq formalisation i.e. **havoc** will not be taken into account in definitions and theorems. However, we will at appropriate place (page 62) provide a small sketch how **havoc** can be incorporated.

Omitting **havoc** has no insurmountable consequences: during the translation from source language to intermediate verification language, one only has to introduce a fresh variable[1] instead of **havoc**ing an existing one.

6.1 Single Assignment Form

This section describes an algorithm to transform an arbitrary command to single assignment form (SA), a form where each variable is assigned to at most once during execution *and is not read from prior to this assignment*. As noted in Sect. 4.2, all variables have values pre-assigned to them. For example, the program **assert** $x = 3; x := 8$ assigns a value to x twice: the implicit initial binding, and the assignment of 8 to x. Thus, the example program is *not* SA. A SA-transformation would be **assert** $x = 3; y := 8$.

The algorithm presented in this section extends identifiers with version numbers to deal with this issue in a simple way. One can imagine all identifiers in the original program having an implicit version number of 0 associated with them, e.g. **assert** $x_0 = 3; x_0 := 8$. Transforming this into SA then becomes a matter of simply incrementing the version number in each assignment: **assert** $x_0 = 3; x_1 := 8$. Similarly, all expressions need to be updated so they refer to the correct version of variables. Thus, using version numbers is an easy way to achieve “variable freshness.” To keep track of variable versions, we use a version map.

Definition 6.1.1 (version map). *A* version map $v : \mathcal{V}$ *is a total function from identifiers to natural numbers.*

$$\mathcal{V} \equiv \mathrm{id} \to \mathbb{N}$$

The transformation algorithm (see Def. 6.1.6) then takes a command and version map, and returns a transformed command and a new version map, e.g.

$$\mathrm{SA}(\textbf{assert}\ x = 3; x := 8, \lambda\ id\,.\,0) = \\ (\textbf{assert}\ x_0 = 3; x_1 := 8, (\lambda\ id.\,0)[x := 1])$$

[1]Our proposed solution for incorporating **havoc** (page 62) essentially automates this process.

The SA transformation is rather straightforward except for choice commands. It would be tempting to turn the program graph into a tree (e.g. changing $(c_a \; [] \; c_b); c_2$ into $(c_a; c_2) \; [] \; (c_b; c_2)$), which leads to a series of deterministic programs which can be dealt with separately in a simple manner. However, the number of such programs grows exponentially with the number of choice commands, which defeats the original goal of avoiding exponential blowup during VC generation.

A better, more efficient way to deal with choice commands consists of first transforming both branches to SA separately. Since both branches could perform assignments to different variables, their versions could get "out of sync": the algorithm returns different version maps for either branch.

Example 6.1.1. *Consider the following example:*

$$x := 0; y := 0; (x := 5 \; [] \; y := 5); \textbf{assert} \; x \neq y$$

A first attempt to transform this to SA could be

$$x_1 := 0; y_1 := 0; (x_2 := 5 \; [] \; y_2 := 5); \textbf{assert} \; x_2 \neq y_2$$

This is clearly not correct: if the left path is taken, x_2 should be compared to y_1, and conversely, if the right path is taken, x_1 should be compared to y_2.

To solve this problem, we build a "target version map" v_t which both branches have to accommodate to: at the end of both their executions, all variables should have the versions mentioned in v_t.

Example 6.1.2. *To continue with Example 6.1.1, taking $v_t = (\lambda \; id \, . \, 0)[x \mapsto 2][y \mapsto 2]$, we get*

$$\ldots; (x_2 := 5; y_2 := y_1 \; [] \; y_2 := 5; x_2 := x_1); \textbf{assert} \; x_2 \neq y_2$$

The $y_2 := y_1$ and $x_2 := x_1$ commands can be seen as synchronisation commands, which transform the current version map to the target version map.

In order to transform $c_a \; [] \; c_b$ with initial version map v_0, the actual algorithm proceeds as follows: it first transforms both branches c_a and c_b normally, resulting in transformed commands c'_a and c'_b and updated version maps v_a and v_b. It then builds (Def. 6.1.4) a target version map by taking the maximum version number for each variable: $v_t(x) = \max(v_a(x), v_b(x))$. Next, it creates synchronisation commands d_1 and d_2 (Def. 6.1.5) to handle the version transition from v_a and v_b to v_t. The final result of the transformation of the choice command is then $(c'_1; d_1 \; [] \; c'_2; d_2, v_t)$.

Definition 6.1.2 (`EWP.version_expr`)**.** *The versioning $[\![e]\!]_v$ of an expression e with respect to a version map v is defined as the following expression:*

$$[\![e]\!]_v \equiv \lambda \; \mu. \; e(\lambda \; x. \; \mu(x_{v(x)}))$$

Definition 6.1.3 (`EWP.targets`). *The function* targets : command $\to \mathcal{P}(\text{id})$ *collects all assignment targets in a command.*

$$\begin{aligned}
\text{targets}(\mathbf{skip}) &= \emptyset \\
\text{targets}(\mathbf{assert}\ e) &= \emptyset \\
\text{targets}(\mathbf{assume}\ e) &= \emptyset \\
\text{targets}(x{:=}e) &= \{x\} \\
\text{targets}(c_1; c_2) &= \text{targets}(c_1) \cup \text{targets}(c_2) \\
\text{targets}(c_1\ []\ c_2) &= \text{targets}(c_1) \cup \text{targets}(c_2)
\end{aligned}$$

Definition 6.1.4 (`EWP.join`). *The function* join : $\mathcal{V} \to \mathcal{V} \to \mathcal{V}$ *yields a new version map containing the most recent version for each identifier.*

$$\text{join}(v, v') = \lambda\ x.\ \max(v(x), v'(x))$$

Definition 6.1.5 (`EWP.sync_vcommand`). *Given a finite set of (unversioned) identifiers* $I = \{x_1, x_2, \ldots, x_n\}$ *and two version maps* v *and* v', *the* synchronisation command sync(I, v, v') *is built as follows:*

$$\begin{aligned}
c_0 &= \mathbf{skip} \\
c_i &= \text{if } v(x_i) = v'(x_i) \text{ then } c_{i-1} \text{ else } (x_i)_{v'(x_i)} := (x_i)_{v(x_i)}; c_{i-1} \\
\text{sync}(I, v, v') &= c_n
\end{aligned}$$

Definition 6.1.6 (`EWP.transform_sa`). *The* single assignment transformation *function* SA : command $\to \mathcal{V} \to$ vcommand $\times\, \mathcal{V}$ *transforms a command into an equivalent single assignment command. Figure 6.2 shows the full definition.*

An important property of the single assignment transformation is the preservation of failure: if the original program fails, so will its SA-transformation.

Definition 6.1.7 (`EWP.store_sync_vstore`). *We say stores are* synchronised *with respect to a version map* v *when*

$$\mu \sim^{v} \mu_{\text{v}} \quad \equiv \quad \forall\ x.\ \mu(x) = \mu_{\text{v}}(x_{v(x)})$$

Lemma 6.1.1 (`EWP.sa_transformation_skip`). *Given a command* c *and let* $(c_{\text{sa}}, v') =$ SA(c, v). *If*

$$\langle c, \mu\rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu'\rangle \quad \wedge \quad \mu \sim^{v} \mu_{\text{sa}}$$

then there exists a μ'_{sa} *such that*

$$\langle c_{\text{sa}}, \mu_{\text{sa}}\rangle \longrightarrow^{*} \langle \mathbf{skip}, \mu'_{\text{sa}}\rangle \quad \wedge \quad \mu' \sim^{v} \mu'_{\text{sa}}$$

Proof. By structural induction on c. □

$$\begin{array}{rcl}
\mathrm{SA}(\mathbf{skip}, v) & = & (\ \mathbf{skip}, v\ ) \\[1em]
\mathrm{SA}(\mathbf{assert}\ e, v) & = & (\ \mathbf{assert}\ [\![e]\!]_v, v\ ) \\[1em]
\mathrm{SA}(\mathbf{assume}\ e, v) & = & (\ \mathbf{assume}\ [\![e]\!]_v, v\ ) \\[1em]
\mathrm{SA}(x := e, v) & = & (\ x_{v(x)+1} := [\![e]\!]_v, v[x := v(x)+1]\ ) \\[1em]
\mathrm{SA}(c; c', v) & = & \begin{array}{l} \text{let } (c_{\mathrm{sa}}, v') = \mathrm{SA}(c, v) \text{ in} \\ \text{let } (c'_{\mathrm{sa}}, v'') = \mathrm{SA}(c', v') \text{ in} \\ \quad (c_{\mathrm{sa}}; c'_{\mathrm{sa}}, v'') \end{array} \\[2em]
\mathrm{SA}(c\ []\ c', v) & = & \begin{array}{lcll} \text{let } (c_{\mathrm{sa}}, v'_1) & = & \mathrm{SA}(c, v) & \text{in} \\ \text{let } (c'_{\mathrm{sa}}, v'_2) & = & \mathrm{SA}(c', v) & \text{in} \\ \text{let } t_1 & = & \mathrm{targets}(c) & \text{in} \\ \text{let } t_2 & = & \mathrm{targets}(c') & \text{in} \\ \text{let } v' & = & \mathrm{join}(v'_1, v'_2) & \text{in} \\ \text{let } t & = & t_1 \cup t_2 & \text{in} \\ \text{let } d_1 & = & \mathrm{sync}(t, v'_1, v') & \text{in} \\ \text{let } d_2 & = & \mathrm{sync}(t, v'_2, v') & \text{in} \\ \multicolumn{4}{l}{\quad (\ (c_{\mathrm{sa}}; d_1)\ []\ (c'_{\mathrm{sa}}; d_2), v'\ )} \end{array}
\end{array}$$

Figure 6.2: Single Assignment Transformation Algorithm

Theorem 6.1.1 (EWP.sa_transformation_fail)**.** *Given a command c and* $(c_{\mathrm{sa}}, v') = \mathrm{SA}(c, v)$*,*

$$\langle c, \mu\rangle \longrightarrow^* \text{failure } \mu' \quad \Rightarrow \quad \mu \sim^v \mu_{\mathrm{sa}} \quad \Rightarrow \quad \exists\ \mu'_{\mathrm{sa}}.\ \langle c_{\mathrm{sa}}, \mu_{\mathrm{sa}}\rangle \longrightarrow^* \text{failure } \mu'_{\mathrm{sa}}$$

Proof. By structural induction on c, relying on Lemma 6.1.1 for the sequencing case. □

Lastly, as promised, let us briefly examine how to deal with **havoc** x. The command nondeterministically assigns a value to x, which is similar to have the rest of the program refer to a fresh variable instead of x. This is easily accomplished using the SA transformation:

$$\mathrm{SA}(\mathbf{havoc}\ x, v) = (\mathbf{skip}, v[x := v(x) + 1])$$

This rids us of **havoc** commands and hence we do not need consider it any longer during the following sections.

6.2 Passification

The SA transformation discussed in the previous section still produces assignments (in fact, it even adds some). In order to produce our efficient weakest preconditions, we need to get rid of those. This is where passification comes in: this second transformation rewrites all assignments as assumptions.

Example 6.2.1. *The following example program clearly fails:*

$$x := 1; x := x + 1; \textbf{assert } x = 3$$

First it is transformed into SA:

$$x_1 := 1; x_2 := x_1 + 1; \textbf{assert } x_2 = 3$$

Passification turns this into

$$\textbf{assume } x_1 = 1; \textbf{assume } x_2 = x_1 + 1; \textbf{assert } x_2 = 3$$

Note the importance of the SA-transformation: without it we would get

$$\textbf{assume } x = 1; \textbf{assume } x = x + 1; \textbf{assert } x = 3$$

which would not fail, contrary to the first three programs.

Definition 6.2.1 (`EWP.passify`)**.** Passification *transforms all assignments into equivalent assumptions.*

$$\begin{array}{rcl}
\text{passify}(\textbf{assert } e) & = & \textbf{assert } e \\
\text{passify}(\textbf{assume } e) & = & \textbf{assume } e \\
\text{passify}(\textbf{skip}) & = & \textbf{skip} \\
\text{passify}(c_{\text{sa}}; c'_{\text{sa}}) & = & \text{passify}(c_{\text{sa}}); \text{passify}(c'_{\text{sa}}) \\
\text{passify}(c_{\text{sa}} \ [] \ c'_{\text{sa}}) & = & \text{passify}(c_{\text{sa}}) \ [] \ \text{passify}(c'_{\text{sa}}) \\
\text{passify}(x := e) & = & \textbf{assume } (x = e)
\end{array}$$

The passification algorithm is rather straightforward. As with the SA-transformation, preservation of failure is an important property, i.e. if an SA-program fails, so does its passified form. We prove it formally.

Definition 6.2.2 (`EWP.stores_veq`)**.** *We define* store equivalence up to a certain version map *as*

$$\mu \overset{\leq v}{\sim} \mu' \quad \equiv \quad \forall\, x, n.\ n \leq v(x) \ \Rightarrow\ \mu(x_n) = \mu'(x_n)$$

Lemma 6.2.1 (`EWP.vmultistep_pmultistep_skip`). *Let* $(c_{\text{sa}}, \nu') = \text{SA}(c, \nu)$ *and* $c_{\text{p}} = \text{passify}(c_{\text{sa}})$*, then*

$$\langle c_{\text{sa}}, \mu\rangle \longrightarrow^* \langle \textbf{skip}, \mu'\rangle \quad \Rightarrow \quad \mu' \overset{\leq \nu'}{\sim} \mu'' \quad \Rightarrow \quad \langle c_{\text{p}}, \mu''\rangle \longrightarrow^* \langle \textbf{skip}, \mu''\rangle$$

Proof. By structural induction on c. □

Lemma 6.2.2 (`EWP.single_assignment_monotonic_store_fail`). *Let*

$$(c_{\text{sa}}, \nu') = \text{SA}(c, \nu)$$

then

$$\langle c_{\text{sa}}, \mu\rangle \longrightarrow^* \textsf{failure}\ \mu' \ \Rightarrow\ \mu \overset{\leq \nu}{\sim} \mu'$$

Proof. By structural induction on c. □

Theorem 6.2.1 (`EWP.vmultistep_pmultistep_fail`). *Let*

$$(c_{\text{sa}}, \nu') = \text{SA}(c, \nu) \qquad c_{\text{p}} = \text{passify}(c_{\text{sa}})$$

then

$$\langle c_{\text{sa}}, \mu\rangle \longrightarrow^* \textsf{failure}\ \mu' \quad \Rightarrow \quad \langle c_{\text{p}}, \mu'\rangle \longrightarrow^* \textsf{failure}\ \mu'$$

Proof. By structural induction on c. The sequence case depends on Lemma 6.2.1 and Lemma 6.2.2. □

6.3 Efficient Weakest Preconditions

We are now ready to define a version of the weakest precondition algorithm optimised for passive programs [52, 77]. As mentioned before, our goal is to avoid the duplication of Q when dealing with choice commands (see Def. 5.4.1):

$$\text{wp}(c_1\ []\ c_2, Q) = \lambda\ \mu.\ \text{wp}(c_1, Q)(\mu) \wedge \text{wp}(c_2, Q)(\mu)$$

The first step consists of making use of Theorem 5.4.3:

$$\text{wp}(c, Q)(\mu) \iff \text{wlp}(c, Q)(\mu) \wedge \text{wp}(c, \text{true})(\mu)$$

This results in the following new expression for choice:

$$\text{wp}(c_1\ []\ c_2, Q)(\mu) = \lambda\ \mu.\ \text{wp}(c_1, \text{true})(\mu) \wedge \text{wp}(c_2, \text{true})(\mu) \wedge \text{wlp}(c_1\ []\ c_2, Q)(\mu)$$

However, this merely moves the problem elsewhere, as the weakest liberal preconditions algorithm also duplicates Q (see Def. 5.3.1):

$$\text{wlp}(c_1 \;[]\; c_2, Q) = \lambda\ \mu.\ \text{wlp}(c_1, Q)(\mu) \wedge \text{wlp}(c_2, Q)(\mu)$$

Until now, we have not made use of the fact that our programs are passive. We know that execution starting in an initial state satisfying $\text{wlp}(c, Q)$ will end in a state satisfying the postcondition Q. Since the state remains unchanged in the case of passified programs, Q must already have been satisfied by the initial program state. In simplified words, if we want to end up in Q, we only need to start in Q. Equating $\text{wlp}(c, Q)$ to Q is too strict however: all states for which execution does not reach the end of the program must also be allowed. For example, take the program $c = \textbf{assume}\ (x > 0)$ and $Q = (x = 5)$. While all states where $x = 5$ must clearly be admitted by $\text{wlp}(c, Q)$, so must all states where $x \leq 0$. We now make use of this information through the following theorem:

Lemma 6.3.1 (`EWP.wlp_rewrite`)**.** *For any passive program* c_p*,*

$$\forall\ \mu.\ \text{wlp}(c_\text{p}, Q)(\mu) \iff \text{wlp}(c_\text{p}, \text{false})(\mu) \vee Q(\mu)$$

Proof. By structural induction on c_p. We rely on Lemma 5.4.4 and the relation $Q(\mu) \Rightarrow \text{wlp}(c_\text{p}, Q)(\mu)$ holds for passified programs (proven in `EWP.Q_impl_wlpQ`). □

This lemma allows us to reformulate the weakest liberal preconditions as follows:

$$\text{wlp}(c_\text{p} \;[]\; c'_\text{p}, Q) = \lambda\ \mu.\ \big(\text{wlp}(c_\text{p}, \text{false})(\mu) \wedge \text{wlp}(c'_\text{p}, \text{false})(\mu)\big)\ \vee\ Q(\mu)$$

In summary, these are the definitions for the efficient weakest preconditions and efficient weakest liberal preconditions:

Definition 6.3.1 (`EWP.efficient_wlp`)**.** *The* efficient weakest liberal precondition $\text{wlp}_\text{e}(c_\text{p}, Q)$ *of a passive command* c_p *with respect to a postcondition* Q *is defined as*

$$\begin{array}{rcl}
\text{wlp}_\text{e}(\textbf{assert}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \Rightarrow Q(\mu) \\
\text{wlp}_\text{e}(\textbf{assume}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \Rightarrow Q(\mu) \\
\text{wlp}_\text{e}(\textbf{skip}, Q) & = & Q \\
\text{wlp}_\text{e}(c_\text{p}; c'_\text{p}, Q) & = & \lambda\ \mu.\ \text{wlp}_\text{e}(c_\text{p}, \text{wlp}_\text{e}(c'_\text{p}, Q))(\mu) \\
\text{wlp}_\text{e}(c_\text{p} \;[]\; c'_\text{p}, Q) & = & \lambda\ \mu.\ (\text{wlp}_\text{e}(c_\text{p}, \text{false})(\mu) \wedge \text{wlp}_\text{e}(c'_\text{p}, \text{false})(\mu)) \vee Q
\end{array}$$

Definition 6.3.2 (`EWP.efficient_wp`)**.** *The* efficient weakest precondition *of a passive command* c_p *with respect to a postcondition* Q*, denoted* $\text{wp}_\text{e}(c_\text{p}, Q)$*, is defined* *as*

$$\begin{array}{rcl}
\mathrm{wp_e}(\mathbf{assert}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \land Q \\
\mathrm{wp_e}(\mathbf{assume}\ e, Q) & = & \lambda\ \mu.\ e(\mu) \Rightarrow Q \\
\mathrm{wp_e}(\mathbf{skip}, Q) & = & Q \\
\mathrm{wp_e}(c_\mathrm{p}; c'_\mathrm{p}, Q) & = & \lambda\ \mu.\ \mathrm{wp_e}(c_\mathrm{p}, \mathrm{wp_e}(c'_\mathrm{p}, Q))(\mu) \\
\mathrm{wp_e}(c_\mathrm{p}\ []\ c'_\mathrm{p}, Q) & = & \lambda\ \mu.\ \ \mathrm{wp_e}(c_\mathrm{p}, \mathrm{true})(\mu) \land \mathrm{wp_e}(c'_\mathrm{p}, \mathrm{true})(\mu) \land \\
 & & \qquad\quad \mathrm{wlp_e}(c_\mathrm{p}\ []\ c'_\mathrm{p}, Q)(\mu)
\end{array}$$

Theorem 6.3.1 (`EWP.soundness_efficient_wp`). *Soundness of efficient weakest preconditions: for any passive program c_p,*

$$\forall\ \mu.\ \mathrm{wp_e}(c_\mathrm{p}, \mathrm{true})(\mu) \Rightarrow \langle c_\mathrm{p}, \mu\rangle\ \text{succeeds}$$

Proof. Follows from Theorem 5.4.3 and Lemma 6.3.1. □

6.4 Soundness and Size

We are now ready to prove the soundness of the approach described in the previous sections, namely transforming into single assignment form, passifying and deriving verification conditions using the efficient weakest precondition algorithm. For our purposes, it is sufficient to prove that given an arbitrary program c we can verify whether or not execution will encounter failure. If we are interested in enforcing specific postconditions or interested in allowing certain preconditions, it is possible to add these as assertions at the end or assumptions at the beginning of the program, respectively.

Theorem 6.4.1. *Given an arbitrary program c, and let c_p be its passified SA-form, then*[2]

$$\forall\ \mu.\ \mathrm{wp_e}(c_\mathrm{p}, \mathrm{true})(\mu) \Rightarrow \langle c, \mu\rangle\ \textit{succeeds}$$

Proof. By chaining Theorem 6.1.1, Theorem 6.2.1 and Theorem 6.3.1 together. In sketch form:

[2]Note Theorem 6.3.1 states executing the *passified* form will not fail, while this theorem states that the *original* command will not fail.

$$\begin{array}{rcll}
\text{original fails} & \Rightarrow & \text{SA form fails} & \text{Theorem 6.1.1} \\
 & + & & \\
\text{SA form fails} & \Rightarrow & \text{passified SA form fails} & \text{Theorem 6.2.1} \\
 & \Downarrow & & \\
\text{original fails} & \Rightarrow & \text{passified SA form fails} & \\
 & \Downarrow & & \text{modus tollens} \\
\text{passified form succeeds} & \Rightarrow & \text{original succeeds} & \\
 & + & & \\
\text{efficient wp hold} & \Rightarrow & \text{passified form succeeds} & \text{Theorem 6.3.1} \\
 & \Downarrow & & \\
\text{efficient wp hold} & \Rightarrow & \text{original succeeds} &
\end{array}$$

□

In order to prove that the efficient weakest preconditions are polynomial in size, we need to view the store predicates as syntactic entities instead of functions, on which we can then define a metric.

Definition 6.4.1 (`EWP.formula_metric`). *We define the following metric on state predicates:*

$$\begin{array}{rcl}
|P \wedge P'| & = & |P| + |P'| + 1 \\
|P \vee P'| & = & |P| + |P'| + 1 \\
|P \Rightarrow P'| & = & |P| + |P'| + 1 \\
|atom| & = & 1
\end{array}$$

Definition 6.4.2 (`EWP.command_metric`). *We define a metric on commands as follows:*

$$\begin{array}{rcl}
|\mathbf{skip}| & = & 1 \\
|\mathbf{assert}\ e| & = & 2 \\
|\mathbf{assume}\ e| & = & 2 \\
|c_1; c_2| & = & |c_1| + |c_2| + 1 \\
|c_1\ [\,]\ c_2| & = & |c_1| + |c_2| + 1
\end{array}$$

Using these metrics, we can express that the verification condition grows quartically with respect to the command size. Note that although wp_e produces a formula which grows quadratically, it operates on the *passified* form of the command. Only the size of the verification conditions with respect to the command *in its original form* is meaningful to us.

Theorem 6.4.2 (`EWP.polynomial_wps`)**.** *The verification condition generated by the efficient weakest precondition algorithm produces a formula which is quartic with respect to the original command size. For any command c and its passified form c_p,*

$$|\mathrm{wp}_e(c_p, Q)| = O(|c|^4 + |Q|)$$

Proof. The verification condition generation algorithm occurs in three steps: SA transformation, passification and verification condition generation. Each phase influences the size of the end result.

- The SA transformation produces a new command which grows quadratically with respect to the original command (`EWP.quadratic_sa_transformation`):
$$|\mathrm{SA}(c, v)| = O(|c|^2)$$
- Passification maintains the size (`EWP.passify_maintains_size`):
$$|\mathrm{passify}(c)| = |c|$$
- The efficient weakest precondition algorithm generates a condition which grows quadratically with respect to command size (proved as `EWP.quadratic_wp'`):
$$|\mathrm{wp}_e(c_p, Q)| = O(|c|^2 + |Q|)$$

Taken together, we get $O(|c|^4 + |Q|)$. □

6.5 Conclusion

Chapter 5 discussed three verification condition generation algorithms, namely the strongest postcondition, the weakest liberal precondition and the weakest precondition generation algorithms. For the purposes of verification, the weakest precondition algorithm proved the most useful: contrary to the other two, it can be used to prevent failures. Unfortunately, the verification condition it generates grows exponentially with respect to program size.

This chapter provided a solution to this problem by reformulating the weakest preconditions such that they only grow polynomially in size. However, this alternative form is only valid for passive programs, i.e. programs with no side effects. Therefore, we have also defined a way to transform an arbitrary IVL program into passified form. This is accomplished by first transforming it into single assignment form, i.e. an equivalently behaving rewrite of the program where every assignment targets a fresh variable, so that each variable is only assigned to once. Next comes the passification phase, which replaces all assignments by **assume** commands.

Both transformations and the efficient weakest precondition algorithm have been implemented in Coq as executable functions. Fully mechanised soundness proofs are also available.

Part II

Symbolic Execution and Separation Logic

Chapter 7

Introduction

In this second part, we take a fundamentally different approach to verification. Whereas the first part relied on verification condition generation, the following chapters will instead make use of symbolic execution [75]. In simplified terms, this consists of executing the program, checking at each step that no errors have occurred. What consistutes an error is up to us to determine: this can range from simple divisions by zero mistakes to accessing certain data outside a transaction. While conceptually simple, many problems arise, such as how to deal with external input (which is unknown at verification time) or nonterminating programs. Solutions to these issues will be discussed in detail.

Next to symbolic execution, separation logic [98, 92] will also play a major role in this part. Separation logic is one of the many attempts to solve the frame problem [88, 40], a well known research problem in the field of software verification. Explained briefly, the frame problem is a direct consequence of shared state and consists of the need to specify which memory locations can be affected by an algorithm: whenever dealing with mutable data structures, a change to one object could impact another object due to internal sharing.

For example, a linked list implementation *could* internally have lists share nodes in order to preserve memory. Since this is an implementation detail, this fact remains hidden from clients. However, when verifying client code which deals with two lists, given only the public specification, how is the verifier to know that there is definitely no sharing going on behind the scenes? To remain sound, it needs to take into account the *possibility* of sharing, meaning that changes to one list *could* propagate to the other list.

A linked list implementation generally takes care of keeping lists semantically separated, i.e. changes specifically applied to one list do indeed leave other lists unmodified, even when there is sharing of nodes internally (for example, using copy-on-write whenever required). Hence, client code rightfully assumes changes to a list are localised, yet verification will very probably fail as the verifier insists on also considering situations which do not occur in practice.

To prevent this from happening, it is necessary to define what is generally named a “frame”, i.e. putting upper bounds on the reach a modification operation can have, thus explicitly informing the verifier of what objects could and cannot be impacted. In the case of linked lists, the framing would need to express the fact that list manipulations, such as sorting or appending an element to a list, do only affect that specific list.

Many approaches to framing exist: examples are dynamic frames [70, 71, 72], implicit dynamic frames [104, 106], ownership based methods [30] (such as Universe Types [91, 41] and Dynamic Ownership [79]), and, of course, separation logic [92, 98, 94]. In a nutshell, separation logic can be viewed as Hoare logic [57] extended with spatial connectives, among them the separating conjunction $\star$ which expresses the fact that its operands hold in disjoint parts of the heap. For example, $\text{List}(p) \star \text{List}(q)$ expresses the existence of two nonoverlapping (i.e. not sharing nodes) lists on the heap. As a consequence, changes to one list are guaranteed not to affect the other. The separating conjunction also gives rise to the frame rule. Put intuitively, it allows us to focus on a chosen part of the heap without having to worry about interferences with other parts, thereby freeing us from frame problem related troubles. Whereas most other methods see everything as potentially shared and need proof that data structures do not overlap, separation logic can be said to be taking the inverse approach: by default, everything is separated, and extra measures need to be taken in order to model sharing. A more detailed explanation is given in Sect. 8.1.

Many verifiers rely on separation logic (e.g. Smallfoot [12], jStar [45], Heap-Hop [109], . . .). Part two of this thesis will focus on VeriFast [67], a verifier developed at the KU Leuven. To familiarise readers with this tool, Chapter 8 introduces the tool’s most important features. It relies on many developments in the field of verification, such as symbolic execution [75], separation logic [98] and abstract predicates [94]. While each of its components is sound, we have no such guarantee about the whole.

This brings us to this part’s first contribution, namely Featherweight VeriFast, presented in Chapter 9. It is a formalisation of VeriFast’s core together with a soundness proof. All definitions [110] have also been implemented in Coq [33, 16], resulting in an executable verifier. Proofs, however, have only been partially mechanised. This first contribution has not yet been published.

Our second contribution in this part focuses on automation (Chapter 10). VeriFast is not fully automated but requires help from the programmer: all code to be verified must be annotated. These annotations can take on considerable proportions. While not forming an insurmountable obstacle, it can make verification unnecessarily complex. Automation, i.e. automatic generation of annotations, can alleviate this issue. Chapter 10 discusses a number of automation techniques and how they can be applied without compromising verification soundness. This second contribution has been published in the following paper:

- Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. Annotation inference for separation logic based verifiers. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2011.

Lastly, we conclude with Chapter 11, where we discuss related and future work.

Chapter 8

VeriFast Introduction

VeriFast [67] is a static verification tool, i.e. it can be used to verify correctness properties (for C and Java) ranging from memory safety and absence of race conditions to full functional correctness. It supports both single-threaded and multithreaded [66] code and it has been used in real-world applications [95, 68].

This chapter introduces the reader to VeriFast. The tool has many features, too many to discuss all in detail. As only a small subset is needed for our purposes, we will restrict ourselves to explaining a core part.

First Sect. 8.1 briefly presents separation logic upon which VeriFast relies. Section 8.2 then proceeds to detail the relevant features of VeriFast.

8.1 Separation Logic

An introduction to VeriFast would be incomplete without providing an overview of the essentials of separation logic [98], which is used by VeriFast to model the current program state. We first present the rationale behind separation logic (Sect. 8.1.1), after which, in Sect. 8.1.2, we describe its main features, namely separating conjunction and the frame rule which follows from it.

```
struct list {
  int         item;
  struct list *next;
};
```

Listing 8.1: A Linked List in C

```
struct list *reverse(struct list *xs) {
  struct list *ys = 0;

  while ( xs != 0 ) {
    struct list *zs = xs->next;
    xs->next        = ys;
    ys              = xs;
    xs              = zs;
  }
  return ys;
}
```

Listing 8.2: List Reversal in C

8.1.1 Rationale

Let us take list reversal in C as working example[1]: given a mutable singly-linked list $[x_1, x_2, \ldots, x_n]$, we modify it in-place such that it becomes $[x_n, x_{n-1}, \ldots, x_1]$. The data structure definition and list reversal code are shown in Listing 8.1 and Listing 8.2, respectively. Since we are working in C, we wish to ensure memory safety and verify that no memory leaks occur.

We define a predicate List(p) which represents the fact that a 0-terminated non-cyclic singly-linked list resides on the heap, starting at address p:

$$\begin{array}{rcl}
\mathrm{List}(p) & \equiv & p = 0 \vee (p \neq 0 \wedge \exists\, n, q.\ \begin{array}[t]{l} p \rightarrow \mathrm{item} = n\ \wedge \\ p \rightarrow \mathrm{next} = q\ \wedge \\ \mathrm{List}(q)\ \wedge \\ p \notin \mathrm{reach}(q)) \end{array} \\
\mathrm{reach}(0) & \equiv & \emptyset \\
\mathrm{reach}(p) & \equiv & \{p\}\ \cup\ \mathrm{reach}(p \rightarrow \mathrm{next})
\end{array}$$

The helper function reach(p) collects all memory addresses reachable through next pointers starting at p and is required to prevent the list from being cyclic.

[1]This example is based on [98].

```
struct list* reverse(struct list *xs)
  requires List(xs)
  ensures  List(result)
```

Listing 8.3: Contract for Linked List Reversal C

The contract for our list reversal function is shown in Listing 8.3: the argument must point to a list, and since the head node of the list has changed, it is necessary to return a pointer to the new head, i.e. no guarantees are made about what xs points to.

In order to verify the while loop, we need a loop invariant for which the following traditional rules must be upheld:

1. it must hold when execution reaches the loop;
2. executing the loop body in a state satisfying the invariant must lead to a new state which also satisfies the invariant;
3. the invariant is true when we leave the loop.

From the code, we see that the loop deals with two lists simultaneously: at each step, ys "steals" a node of xs, making ys grow and xs shrink, until nothing remains of xs and ys points to the reversed form of the original list. If we choose the following invariant

$$\mathrm{List}(xs) \wedge \mathrm{List}(ys)$$

the first invariant rule is respected and the third rule results in having the program state immediately satisfy the postcondition after the loop. However, proving that the second rule holds is impossible. This is due to the fact the lists pointed to by xs and ys might share nodes. For example, the loop invariants admits the memory layout shown on the left side of Fig. 8.1. After one iteration, we would end up in the situation as depicted on the right side of the figure, which clearly does not satisfy the invariant as ys points to a cyclic list.

We need to strengthen the invariant by adding the condition that both lists do not share nodes. The reinforced loop invariant is

$$\mathrm{List}(xs) \;\wedge\; \mathrm{List}(ys) \;\wedge\; \mathrm{reach}(xs) \cap \mathrm{reach}(ys) = \emptyset$$

which allows us to deal successfully with the loop. Given this, we can prove that the function does not dereference dangling or wild pointers, and does not leak memory.

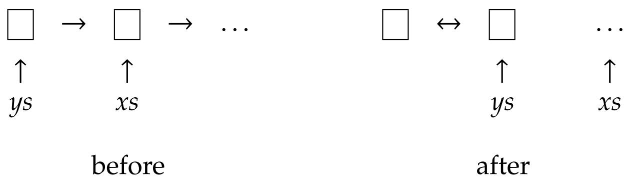

Figure 8.1: Linked Lists Sharing Nodes

```
struct list *xs = ...;
struct list *ys = ...;

xs = reverse(xs);
```

Listing 8.4: Calling the List Reversal Function

Unfortunately, new problems arise when calling the function. Listing 8.4 gives an example: we have two lists xs and ys at our disposition and wish to reverse the first one. The current program state does not match the function's precondition, which only expects one list to be present in memory. We could extend the contract so that it allows for an extra list (or any number of other data structures), in which case we also need to express somehow that the list to be reversed does not share memory with any other data structure, lest the function modifies these in seemingly arbitrary ways. More specifically, we wish that the contract for the reverse function expresses that it will only modify the given list and leaves the rest of the heap untouched.

8.1.2 Separating Conjunction and Frame Rule

Separation logic provides an elegant solution to the problems described in the previous section. We can model the heap as a partial function mapping addresses to memory cell contents, for example $\mathbb{N} \rightharpoonup \mathbb{Z}_{256}$. For simplicity however, we will generally take $\mathbb{N}$ as the range of the heap function, but for our current purposes, this detail does not matter. The domain of the function corresponds to the allocated parts of the heap. Instead of dealing directly with one monolithic heap, separation logic chooses to model the heap as composable subheaps. Let us first define the same notations as used in [98]:

$$\begin{array}{rcll} h \perp h' & \equiv & \text{dom}(h) \cap \text{dom}(h') = \emptyset & \\ h \cdot h' & \equiv & h \cup h' & \text{with } h \perp h' \end{array}$$

That is, $h \perp h'$ states that two heaps are disjoint, and $h \cdot h'$ merges two disjoint subheaps together into a larger heap. Separation logic then extends regular predicate logic with new forms of assertions that aid us in describing the heap succinctly:

- Separating conjunction $P \star Q$ indicates that the heap can be divided into two disjoint subheaps which satisfy P and Q, respectively.

$$h \models P \star Q \quad \Longleftrightarrow \quad \exists\, h_1, h_2.\ h_1 \cdot h_2 = h \ \wedge\ h_1 \models P \ \wedge\ h_2 \models Q$$

- The empty heap assertion **emp** states the heap is empty.

$$h \models \mathbf{emp} \quad \Longleftrightarrow \quad \mathrm{dom}(h) = \emptyset$$

- The singleton heap assertion $a \mapsto b$ states that the heap contains only one allocated cell; it resides at address a and contains the value b.

$$h \models a \mapsto b \quad \Longleftrightarrow \quad \mathrm{dom}(h) = \{a\} \ \wedge\ h(a) = b$$

We now examine how this helps us solve our problems from the previous section. One issue concerned the sharing of memory cells between different data structures, in our example these being the linked list nodes. Separating conjunction allows us to concisely express that no sharing does occur. We redefine our List predicate[2] as follows:

$$\mathrm{List}(p) \equiv p = 0 \vee (p \neq 0 \wedge \exists\, n, q.\ (p \to \mathrm{item}) \mapsto n \ \star\ (p \to \mathrm{next}) \mapsto q \ \star\ \mathrm{List}(q))$$

In this new definition, separating conjunction automatically prevents the list from being cyclic since the tail of the list is guaranteed to reside in a different portion of the heap than the head. We modify the loop invariant in a similar way:

$$\mathrm{List}(xs) \ \star\ \mathrm{List}(ys)$$

Using separating conjunction not only leads to syntactically shorter formulae, it also provides us with a new level of abstraction which considerably simplifies reasoning about the heap. Another advantage of separation logic comes in the form of the frame rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P \star R\}\ c\ \{Q \star R\}}$$

where no free variable in R is modified by c. This expresses that if a program c behaves a certain way, it will behave the same way if the heap is extended by R, from which follows that it will not touch the subheap described by R.

[2]Technically, we should replace $p \to \mathrm{item}$ by p and $p \to \mathrm{next}$ by $p + 1$, i.e. replacing field accesses by offsets, but for the sake of clarity, we prefer to maintain a certain level of abstraction.

The frame rule solves our second problem mentioned in the previous section: given a function contract, we can always adapt it to our needs using the frame rule. In our case, this becomes

$$\frac{\{\text{List}(xs)\}\ \text{ys} = \text{reverse}(xs)\ \{\text{List}(ys)\}}{\{\text{List}(xs) \star \text{List}(zs)\}\ \text{ys} = \text{reverse}(xs)\ \{\text{List}(ys) \star \text{List}(zs)\}}$$

Thus, we can now call the function in a context where there is a second list present on the current heap. Added to this, we are guaranteed that this second list remains untouched.

In summary, separation logic provides us with tools which considerably simplify reasoning about shared mutable data structures.

8.2 VeriFast: Hands On

In this section, we give an informal description of how VeriFast operates. Many details will be deferred until a later chapter; our current goal is to provide context which will aid the reader in understanding the rationale behind the formalisation presented in Chapter 9.

Although VeriFast can handle both C and Java programs, we will focus solely on C programs as using Java merely adds complexity without providing extra insight.

8.2.1 Basics

In order to verify C code with VeriFast, every function needs to be given a contract, i.e. a precondition and a postcondition. VeriFast will then consider every function in turn, checking that it obeys its contract. More specifically, VeriFast will set up an arbitrary program state which satisfies the precondition, (symbolically) execute [75] the function body, and verify that the final program state satisfies the postcondition.

The program state consists of a store (keeping track of the values of local variables) and a heap[3]. The heap is abstractly represented by a multiset of *heap chunks*, each of which represents the presence of a certain data structure residing in its own part of the heap. In other words, the heap chunks are implicitly conjoined using the separating conjunction. In Sect. 8.1.2, we already

[3] A third component is the path condition, but we will ignore this for now.

encountered one kind of heap chunk, namely the $a \mapsto b$ chunk, which expresses the existence of an allocated memory cell at address a containing the value b.

Example 8.2.1. *Consider the heap with a two allocated memory cells, at location* 3 *and* 7*, initialised to* 4 *and* 8*, respectively. This heap can be represented as a partial function h as follows:*

$$\mathrm{dom}(h) = \{3, 7\} \qquad h(3) = 4 \qquad h(4) = 8$$

Using a separation logic formula, we can model the same heap with

$$3 \mapsto 4 \star 7 \mapsto 8$$

This formula unambiguously describes h: h is the only heap which is a model of this formula:

$$\forall\, h'. \quad h' \models 3 \mapsto 4 \star 7 \mapsto 8 \quad \Rightarrow \quad h = h'$$

VeriFast expresses the heap as the following multiset:

$$\{\, 3 \mapsto 4, 7 \mapsto 8 \,\}$$

If a pointer p points to the first cell, the store would map p to 3*. The full program state would thus be*

$$\langle s[p := 3], \{3 \mapsto 4, 7 \mapsto 8\}\rangle$$

where s can be an arbitrary store.

Example 8.2.2. *In the previous example, separating conjunction does not help much as it is clear from the addresses that the chunks inhabit different parts of the heap. If, however, we work with symbols (to be explained in detail later), the separating conjunction does provide us with extra information:*

$$p \mapsto v \star p' \mapsto v'$$

implies that $p \neq p'$*. This formula has as models all heaps with exactly two allocated memory cells:*

$$h \models p \mapsto v \star p' \mapsto v'$$
$$\Updownarrow$$
$$\exists\, p, p', v, v'.\; p \neq p' \wedge \mathrm{dom}(h) = \{p, p'\} \wedge h(p) = v \wedge h(p') = v'$$

Execution consists of having every statement transform the program state in a predefined and generally straightforward way: assignments to variables update the store, dereferencing a pointer looks up the appropriate chunk in the current heap, etc. Allocation (malloc) is worthy of closer examination, as it is used ubiquitously and introduces us to both of VeriFasts built-in heap chunks.

Example 8.2.3. *Consider again the C structure defined in Listing 8.1 on page 78. The following statement*

```
struct list *p = malloc( sizeof( struct list ) );
```

allocates a new data structure on the heap. The memory location is picked at random, let's call it ℓ. The store is updated so that ptr is bound to ℓ. VeriFast also generates three heap chunks:

- $(\ell \rightarrow \text{item}) \mapsto a$ *where a is fresh with respect to the current program state. This chunk expresses that memory has been allocated at address $\ell \rightarrow$ item and contains some value a of which nothing is known. This accurately models the way allocation works in C: no guarantees are made about the initial contents of freshly allocated memory. If, later on, a statement attempts to read from or write to this memory location, VeriFast requires this chunk to be present on the heap, otherwise verification fails.*
- $(\ell \rightarrow \text{next}) \mapsto b$ *where b is fresh with respect to the current program state. The same applies as with $(\ell \rightarrow \text{item}) \mapsto a$.*
- *malloc_block_list(ℓ) is more of a "virtual" chunk as its main reason is to keep track of typing information: in case a pointer is cast to another type, VeriFast needs to remember its original type, so as to know which cells to deallocate when the pointer is freed.*

Freeing the list node, i.e. free(p); does the converse: it looks up in the store to what value p is bound and removes the same three chunks as mentioned above. Freeing while the current heap does not contain all of these chunks is considered an error by VeriFast, as it (possibly[4]) corresponds to freeing unallocated memory.

8.2.2 User Defined Predicates

Consider the cell structure shown in Listing 8.5. A function for deallocating cells is shown in Listing 8.6. Similarly to what we explained before, in order to free a cell, we need both the malloc_block_cell(p) and p–>contents $\mapsto v$ chunks representing the fact that there actually is a cell structure at the memory location p points to. Therefore, we must express this requirement in destroy's precondition. The precondition in Listing 8.6 is of course written in VeriFast

[4]At runtime, it is still possible that no error occurs if the necessary chunks were just hidden in some way at verification time. If VeriFast allows the deallocation, it is certain that it is also allowed at runtime. The converse is not necessarily true.

```
struct cell {
  int contents;
};
```

Listing 8.5: Cell Struct

```
void destroy(struct cell *p)
//@ requires malloc_block_cell(p) &*& p->contents|->?v;
//@ ensures emp;
{
  free(p);
}
```

Listing 8.6: Cell Destruction

syntax and corresponds to

$$\exists\, v.\ \text{malloc\_block\_cell}(p) \star p\text{–>contents} \mapsto v$$

i.e. VeriFast denotes existentials with question marks and separating conjunction is written &*&. VeriFast annotations are always put inside comments so that the source remains valid C code.

Note how the postcondition describes the empty heap: this mirrors the fact that after calling the function, the subheap originally containing the cell has been reduced to the empty heap, or, in other words, the memory occupied by the cell has been freed. Conversely, if the cell would have remained in memory, we would need to explicitly mention this in the postcondition, as shown in Listing 8.7.

Whenever a function needs to interact with cells, we would need to mention the appropriate chunks in the precondition and/or postcondition. The situation is exacerbated if the structures contain multiple fields, thereby increasing the number of chunks to be explicitly written down in the contracts. To remedy

```
void donothing(struct cell *p)
//@ requires malloc_block_cell(p) &*& p->contents|->?v;
//@ ensures  malloc_block_cell(p) &*& p->contents|->v;
{ /* NOP */ }
```

Listing 8.7: Elaborately Leaving a Cell Untouched

```
predicate Cell(struct cell *p) =
  malloc_block_cell(p) &*& p->contents |-> ?v;
```

Listing 8.8: A User Defined Predicate for Cells

```
void destroy(struct cell *p)
  //@ requires Cell(p);
  //@ ensures  emp;
{
  free(p);
}
```

Listing 8.9: Updated Contract for destroy

this, VeriFast provides user defined predicates [94], which allow us to wrap multiple chunks together in a single chunk. In the case of cells, we could define the predicate shown in Listing 8.8. This allows us to rewrite destroy's contract more abstractly, as shown in Listing 8.9.

However, verification fails when using this new contract. The reason is simple: the free statement expects to see the malloc_block_cell(p) and p–>contents $\mapsto v$ chunks on the heap, but it can only find a Cell(p) chunk, which originates from the precondition. While we know that this chunk is just an abstraction for the two chunks, VeriFast will not automatically "unfold" it into its two constituents. We need to perform this unfolding manually, and for this VeriFast provides the **open** statement. Listing 8.10 illustrates its use. There also exists its complementary statement **close**, which does the opposite of **open**.

```
void destroy(struct cell *p)
  //@ requires Cell(p);
  //@ ensures  emp;
{
  //@ open Cell(p);
  free(p);
}
```

Listing 8.10: Corrected Version of destroy

```
void triple(struct cell *p)
  //@ requires Cell(p);
  //@ ensures  Cell(p);
{
  //@ open Cell(p);
  int contents = p->contents;
  p->contents  = 3 * contents;
  //@ close Cell(p);
}
```

Listing 8.11: Memory Safe Triple

The **open** Cell(p) statement requires a Cell chunk on the heap and replaces it by the predicate's body, as defined in Listing 8.8:

$$\{\text{Cell}(p)\}\ \textbf{open}\ \text{Cell}(p)\ \{\text{malloc\_block\_cell}(p) \star p\text{–>contents} \mapsto v\}$$

After the **open** statement has exposed the appropriate chunks, free is able to perform its duty, i.e. removing both chunks from the heap, resulting in an empty heap, which matches the postcondition. Thus, destroy has been successfully verified.

The Cell predicate in its current state throws information away with respect to its constituents. For example,

$$\begin{gathered}\{\ \text{malloc\_block\_cell}(p) \star p\text{–>contents} \mapsto 5\}\\ \textbf{close}\ \mathit{Cell}(p)\\ \{\ \text{Cell}(p)\ \}\\ \textbf{open}\ \mathit{Cell}(p)\\ \{\ \text{malloc\_block\_cell}(p) \star p\text{–>contents} \mapsto v\}\end{gathered}$$

The **close** statement first folds the two chunks together into a single Cell chunk, after which **open** unfolds it back. However, the information about the cell's contents has been lost: initially it was known to be 5, but the final program state does not remember this, i.e. it only states the cell contains some unknown value v.

If we currently wish to define a function which triples the contents of a cell, we will at best be able to prove it is memory safe and does not deallocate the cell, as shown in Listing 8.11.

Luckily, it is possible to also prove full functional correctness, either by not making use of abstraction (which would be an unfortunate concession on our

```
predicate Cell(struct cell *p, int v) =
  malloc_block_cell(p) &*& p->contents |-> v;
```

Listing 8.12: Improved Cell Predicate

```
void destroy(struct cell *p)
  //@ requires Cell(p, ?v);
  //@ ensures  emp;
{
  //@ open Cell(p, v);
  free(p);
}
```

Listing 8.13: destroy with Improved Cell Predicate

part), or update the Cell predicate so that it keeps track of the cell's value. We do this by ridding ourselves of the existential ?*v* and turn it into an extra argument, as shown in Listing 8.12. An updated version for destroy can be found in Listing 8.13, but it does not offer any advantages with respect to its previous version in Listing 8.10.

Listing 8.14 shows an improved version of the triple function. Note how we can use wildcards arguments (i.e. _), in which case VeriFast will automatically infer[5] the correct values for us.

[5]The usage of wildmarks can also lead to ambiguities; how these are dealt with will be discussed later.

```
void triple(struct cell *p)
  //@ requires Cell(p, ?v);
  //@ ensures  Cell(p, ?w) &*& w == 3 * v;
{
  //@ open Cell(p, _);
  int contents = p->contents;
  p->contents  = 3 * contents;
  //@ close Cell(p, _);
}
```

Listing 8.14: triple Proven Fully Functional Correct

```
predicate Node(struct list *p, int v, struct list* q) =
  malloc_block_list(p) &*&
  p->item |-> v       &*&
  p->next |-> q;

predicate List(struct list *p) =
  p == 0 ? emp : Node(p, _, ?next) &*& List(next);
```

Listing 8.15: List Predicate

8.2.3 Recursive User Defined Predicates

In the previous section, we discussed user defined predicates. From our explanations, the reader might have the impression that user defined predicates only add a level of abstraction which merely simplifies interacting with chunks. In this section, we show how they do actually make VeriFast more expressive.

We will again use singly linked lists as example (see Listing 8.1 on page 78). The length of linked lists is generally not known at verification time; instead, programs usually jump from node to node until a null pointer is found, which indicates the end of the list. If we are to write a function which deallocates all memory used by a linked list, what would its contract be? Using only the primitive chunks (malloc_block_xxx(p) and $p \mapsto v$), we cannot describe linked lists of arbitrary length.

User defined predicates allow us to define a recursive list predicate as shown in Listing 8.15. We have introduced two layers of abstractions: a Node(p, v, n) represents a a single linked list node on the heap and keeps track of the values of both fields, so it consists of three chunks. List(p) builds the actual linked list chain: either $p = 0$, and the list is empty, or $p \neq 0$, and it has a head (a node) and a tail (another list).

Let us manually step through the code shown in Listing 8.16. The contract is straightforward and expresses that it deallocates a linked list pointed to by xs. The first statement unfolds the List(xs) predicate. Because its definition contains a conditional, a case split occurs, i.e. execution forks[6] into two different branches where xs = 0 and xs ≠ 0 respectively.

We first consider the case xs = 0. This fact about `xs` is registered in the program state's *path condition* (a series of conditions putting restrictions on the variables'

[6]This forking is not related in any way with multithreading.

```
 1 void destroy_list(struct list *xs)
 2   //@ requires List(xs);
 3   //@ ensures  emp;
 4 {
 5   //@ open List(xs);
 6   if ( xs != 0 )
 7   {
 8     //@ open Node(xs, _, _);
 9     destroy_list(xs->next);
10     free(xs);
11   }
12 }
```

Listing 8.16: Destroying Lists

values). Remembering this fact might prove important later on[7]: for example in case the code has another `if ( xs != 0 )` appearing later on we do not want the verifier to take into account both clauses, but remember that `xs = 0` and only consider the `then` clause.

Unfolding List(xs) yields **emp**, the if-statement (lines 6–11) is skipped, and the postcondition is satisfied; in other words, this path has been successfully verified. We now consider the alternative execution path where xs $\neq$ 0. Again this fact is registered in the path condition. Opening up the List(xs) predicate populates the heap with Node(xs, v, q) and List(q) chunks. On line 8, we open the Node(xs, v, q) node, producing three new chunks malloc_block_list(xs), xs–>item $\mapsto v$ and xs–>next $\mapsto q$.

On line 9, we recursively call destroy_list. Function calls are dealt with not by executing the function's body, but instead by transforming the program state as described by the function's contract. More specifically, we *consume* the precondition and *produce* the postcondition. Consuming the precondition consists of checking that it is satisfied by the current program state and removing all involved heap chunks from the heap. In our case, this means we first check that a chunk List(q) is present; this being the case, it can be removed from the heap. Next, producing the postcondition consists of adding all chunks it mentions to the heap. In our case, no chunks need to be added.

On line 10 we clean up the head node with free(xs), which effectively removes the remaining chunks malloc_block_list(xs), xs–>item $\mapsto v$ and xs–>next $\mapsto q$.

[7]While the path condition is strictly speaking not important for soundness, it is for completeness.

We thus end up with the empty heap, which is allowed by the postcondition: the second execution branch has also been successfully verified.

8.2.4 List Reversal

To come full circle, let us reconsider the list reversal example shown in Listing 8.2 on page 78. First we will show how to verify that the function is memory safe and does not leak memory. Next, we endeavour to prove full functional correctness, which requires quite a bit more work.

Memory Safety

Listing 8.17 shows an list reversal function which can be successfully verified by VeriFast. Assuming VeriFast is sound, the list reversal algorithm has thus been proven memory safe (i.e. no dereferencing wild or dangling pointers) and not to be leaking memory.

We clarify the VeriFast-specific annotations:

- The **close** statement on line 6 adds a List(*ys*) chunk to the heap, so that it satisfies the loop invariant.
- The **open** statements on lines 11–12 expose the chunk required (xs–>next $\mapsto$ q) to manipulate the fields of the structure pointed to by *xs*.
- Lines 17–18 fold the chunks back, so that the iteration ends with a heap containing the two chunks List(*xs*) and List(*ys*), which satisfies the loop invariant.
- When leaving the loop, the heap contains two chunks: List(*xs*) and List(*ys*). Since $xs == 0$, line 21 has the effect of removing the former. This leaves us with a solitary List(*ys*) chunk, which fits the postcondition.

Full Functional Correctness

This section shows how to prove the list reversal algorithm correct, i.e. that it is memory safe, does not leak memory, *and* is functionally correct. Compared to Listing 8.17 (page 92), which only required a Node and List predicate as supplementary definitions with respect to the C code, we have to rely on

```
struct list *reverse(struct list *xs)
  //@ requires List(xs);
  //@ ensures List(result);
{
  struct list *ys = 0;
  //@ close List(ys);

  while ( xs != 0 )
    //@ invariant List(xs) &*& List(ys);
  {
    //@ open List(xs);
    //@ open Node(xs, _, _);
    struct list *zs = xs->next;
    xs->next = ys;
    ys = xs;
    xs = zs;
    //@ close Node(ys, _, _);
    //@ close List(ys);
  }

  //@ open List(xs);
  return ys;
}
```

Listing 8.17: Verified List Reversal (No Full Functional Crrectness)

more advanced features to obtain a version which has been proven to be fully functional correct.

List reversal operates on the lists' contents; therefore, we need an updated List predicate which keeps track of a list's content. The situation is similar to the one encountered in Sect. 8.2.2 where we needed to extend the Cell predicate with an extra argument (Listing 8.8 on page 86 versus Listing 8.12 on page 88) so that information about a cell's content would be preserved.

A cell's content can be represented by a single integer, which is a built-in datatype. In the case of lists, however, we need to define an appropriate data structure ourselves. For this, VeriFast supports inductive data types. Listing 8.18 shows the definition for an inductively defined integer list: a list is either empty (Nil), or a node with some integer h (the head of the list) and a list t (the tail of the list), written Cons(h, t). This definition shows great similarities to the list struct we defined in C: a list is either the null pointer

```
inductive List = Nil
               | Cons(int, List);
```

Listing 8.18: List Inductive Data Type

```
predicate Node(struct list *p, int v, struct list* q) =
  malloc_block_list(p) &*&
  p->item |-> v        &*&
  p->next |-> q;

predicate List(struct list *p, List Xs) =
  p == 0 ? Xs == Nil
         : Node(p, ?head, ?next)   &*&
           List(next, ?tail)       &*&
           Xs == Cons(head, tail);
```

Listing 8.19: Updated List Predicate

(which corresponds to Nil), or a valid object in memory with two fields item and next, which correspond to the head and tail of a Cons, respectively.

The list $[1, 2, 3]$ can thus be represented by Cons(1, Cons(2, Cons(3, Nil))). We need to link this with actual data structures on the heap. This responsibility lies with the List predicate. An updated version of the List predicate is shown in Listing 8.19: a List(p, xs) chunk on the heap indicates that p points to a linked list in memory whose contents is represented by xs. For example, List(p, Cons(1, Nil)) means p–>value == 1 and p–>next == 0.

We now turn our attention to the function's contract: the function expects its argument to point to a list, which contains some elements xs, and returns a new pointer which points to the same list, but with elements reversed. The contract is shown in Listing 8.20. The postcondition relies on the function Reverse to express the fact that the function reverses its argument. Its definition is shown in Listing 8.21.

```
struct list *reverse(struct list *xs)
  //@ requires List(xs, ?Lst);
  //@ ensures List(result, Reverse(Lst));
```

Listing 8.20: Function Contract

```
fixpoint List Reverse(List xs)
{
  switch ( xs ) {
    case Nil:
      return Nil;
    case Cons(x, xs2):
      return Append( Reverse(xs2), Cons(x, Nil) );
  }
}
```

Listing 8.21: Reverse Function

Note how, after having defined C lists and a corresponding inductive type List, we now have a C list reversal function and a corresponding one operating on List.

$$\begin{tikzcd}
\text{VeriFast} & \text{List} \arrow[d, leftrightarrow] & \text{List Reverse(List xs)} \arrow[d, leftrightarrow] \\
\text{C} & \text{struct list*} & \text{struct list* reverse(struct list* xs)}
\end{tikzcd}$$

These corresponding definitions are typical when verifying code with VeriFast. One could say verification consists of programming the same functionality twice, and show that both implementations behave exactly the same. The chance that one makes exactly the same mistake using two vastly different languages (imperative versus purely functional) is very small. Added to this, VeriFast allows us to prove theorems about the purely functional implementation, furthering our confidence about its correctness.

The Reverse function shown in Listing 8.21 depends on another function Append. While it is certainly possible to avoid this, we need an Append function to express the loop invariant in the C reverse function anyway. Its definition is given in Listing 8.22.

Let us take a closer look at the loop invariant. In our previous version (Listing 8.17 on page 92), the invariant was List(xs) $\star$ List(ys). We now also need it to express how the lists' contents are related:

$$\text{List}(xs, ?Xs) \star \text{List}(ys, ?Ys) \star \text{Append}(\text{Reverse}(Ys), Xs) == Lst$$

where *Lst* is bound by the precondition (Listing 8.20) to the original list.

Listing 8.23 shows the fully annotated code. We focus solely on the parts pertinent to proving full functional correctness and take some liberties regarding

```
fixpoint List Append(List xs, List ys)
{
  switch ( xs ) {
    case Nil:
      return ys;
    case Cons(x, xs2):
      return Cons(x, Append(xs2, ys));
  }
}
```

Listing 8.22: Append Function

the technical details. We also introduce the following shorthand notations:

$$\begin{aligned}
[x_1, x_2, \ldots, x_n] &\equiv \text{Cons}(x_1, \text{Cons}(x_2, \ldots, \text{Cons}(x_n, \text{Nil}))) \\
x : xs &\equiv \text{Cons}(x, xs) \\
xs^{\text{R}} &\equiv \text{Reverse}(xs) \\
xs +\!\!+\, ys &\equiv \text{Append}(xs, ys)
\end{aligned}$$

Line 6 is required to satisfy the invariant, which expects two List chunks on the heap. Thus, just before reaching the loop, the heap contains

$$\text{List}(xs, Lst) \star \text{List}(ys, \text{Nil})$$

The third conjunct of the loop invariant is proven automatically and hence does not require any help from our part. Inside the loop, we start with

$$\text{List}(xs, Xs) \star \text{List}(ys, Ys)$$

Since $xs \neq 0$, we know that Xs is a Cons, whose components (head and tail) are of interest to us. Suffice it to say that we bind these to the variables X and $Tail$ on line 15 and 17 respectively; in other words, after binding them, we can say that $Xs == \text{Cons}(X, Tail)$.

After shuffling around pointers (lines 18–20) and folding predicates (lines 21–22), the heap becomes

$$\text{List}(xs, Tail) \star \text{List}(ys, X : Ys)$$

In other words, the head of xs has become the head of ys. From the invariant, we know that

$$Ys^{\text{R}} +\!\!+\, (X : Tail) == Lst$$

Being at the end of the loop, we need to show that the current heap satisfies the invariant. In other words, we need to show that

$$(X : Ys)^{\text{R}} +\!\!+\, Tail == Lst$$

```
struct list *reverse(struct list *xs)
  //@ requires List(xs, ?Lst);
  //@ ensures  List(result, Reverse(Lst));
{
  struct list *ys = 0;
  //@ close List(ys, Nil);

  while ( xs != 0 )
    /*@ invariant List(xs, ?Xs) &*&
                  List(ys, ?Ys) &*&
                  Append(Reverse(Ys), Xs) == Lst;
    @*/
  {
    //@ open List(xs, Xs);
    //@ open Node(xs, ?X, _);
    struct list *zs = xs->next;
    //@ assert List(zs, ?Tail);
    xs->next = ys;
    ys = xs;
    xs = zs;
    //@ close Node(ys, _, _);
    //@ close List(ys, _);
    //@ AppendAssoc( Reverse(Ys), Cons(X, Nil), Tail );
  }

  //@ open List(xs, _);
  //@ AppendNil(Reverse(Ys));
  //@ RevRev(Ys);

  return ys;
}
```

Listing 8.23: Fully Annotated List Reversal Function

```
lemma void AppendAssoc(List Xs, List Ys, List Zs)
  requires true;
  ensures  Append(Xs, Append(Ys, Zs)) ==
           Append(Append(Xs, Ys), Zs);
{
  switch ( Xs ) {
    case Nil:
      return;
    case Cons(X, Xs2):
      AppendAssoc(Xs2, Ys, Zs);
      return;
  }
}
```

Listing 8.24: AppendAssoc Lemma

To prove this, we need to provide VeriFast with a little help. Let us first see how we might prove this equality:

$$\begin{array}{rcl} (X : Ys)^{\mathrm{R}} +\!\!+ \mathit{Tail} & == & (Ys^{\mathrm{R}} +\!\!+ [X]) +\!\!+ \mathit{Tail} \\ & == & Ys^{\mathrm{R}} +\!\!+ ([X] +\!\!+ \mathit{Tail}) \\ & == & Ys^{\mathrm{R}} +\!\!+ (X : \mathit{Tail}) \\ & == & Ys^{\mathrm{R}} +\!\!+ (X : \mathit{Tail}) \\ & == & \mathit{Lst} \end{array}$$

All these equalities are straightforward and can be shown to hold automatically by VeriFast, except for one: the associativity of Append. We somehow need to inform VeriFast that $(Xs +\!\!+ Ys) +\!\!+ Zs == Xs +\!\!+ (Ys +\!\!+ Zs)$. To this end, VeriFast provides lemmas. For example, associativity of Append is proven by Listing 8.24. The details of the proof are not important; only the contract is of interest to us.

We can use this lemma to provide VeriFast with the missing link in the proof that the loop invariant is satisfied. This is done on line 23 of Listing 8.23, which tells VeriFast that $(Ys^{\mathrm{R}} +\!\!+ [X]) +\!\!+ \mathit{Tail} == Ys^{\mathrm{R}} +\!\!+ ([X] +\!\!+ \mathit{Tail})$. This ends the verification of the loop.

Upon exiting the loop, the heap contains $\mathrm{List}(xs, Xs) \star \mathrm{List}(ys, Ys)$ which, since $xs == 0$ can be simplified (line 26) to just $\mathrm{List}(ys, Ys)$. From the invariant, we also know that $Ys^{\mathrm{R}} +\!\!+ \mathrm{Nil} == \mathit{Lst}$. In order to simplify this to $Ys^{\mathrm{R}} == \mathit{Lst}$, we need another lemma, which is given in Listing 8.25 and applied on line 27.

One last step remains: satisfying the postcondition. We are given a heap with $\mathrm{List}(ys, Ys)$ and $Ys^{\mathrm{R}} == \mathit{Lst}$, and we need this to match this with $\mathrm{List}(ys, \mathit{Lst}^{\mathrm{R}})$.

```
lemma void AppendNil(List Xs)
  requires true;
  ensures  Append(Xs, Nil) == Xs;
{
  switch ( Xs ) {
    case Nil:
      return;
    case Cons(X, Xs2):
      AppendNil(Xs2);
  }
}
```

Listing 8.25: AppendNil Lemma

Using the equality, we can rewrite this to List(ys, $(Ys^{\mathrm{R}})^{\mathrm{R}}$), which means we need to prove that $(Ys^{\mathrm{R}})^{\mathrm{R}} == Ys$. The lemma is given in Listing 8.26 and invoked on line 28.

We are now done verifying the reversal function. In conclusion, in order to prove full functional correctness, we had to made use of multiple VeriFast features.

- User defined predicates were used to describe data structures on the heap. For example, we defined a List predicate which we used to indicate the presence of a linked list on the heap.
- We relied on inductive data types to allow us to abstractly model the values represented by the data structures residing on the heap. For example, we defined a List inductive data type.
- Fixpoints let us define functions over these data types. In order to prove full functional correctness, it will generally be necessary to define a corresponding fixpoint for every C function to be verified, and show that both implementations yield equivalent results. For example, we defined Append and Reverse on lists.
- Lemmas were used to prove equivalences so as to allow us to show that certain conditions (preconditions, postconditions, loop invariants, etc.) are indeed met by the current program state. For example, we showed that Reverse(Reverse(Xs)) == Xs.

```
lemma void RevTail(List Xs, int X)
  requires true;
  ensures  Reverse( Append(Xs, Cons(X, Nil)) ) ==
           Cons(X, Reverse(Xs));
{
  switch ( Xs ) {
    case Nil:
      return;
    case Cons(Y, Ys):
      RevTail(Ys, X);
      return;
  }
}

lemma void RevRev(List Xs)
  requires true;
  ensures  Reverse(Reverse(Xs)) == Xs;
{
  switch ( Xs ) {
    case Nil:
      return;
    case Cons(X, Xs2):
      RevTail(Reverse(Xs2), X);
      RevRev(Xs2);
  }
}
```

Listing 8.26: Reverse Lemmas

8.2.5 Ambiguous Matches

Due to its importance later on, we briefly discuss the possibility of ambiguous matches in VeriFast. An ambiguous match occurs when VeriFast needs to find certain chunks on the heap (such as when opening/closing chunks, a function is called, etc.) but, due to incomplete information, multiple chunks are eligible for selection. This is similar to spooky disjunctions in [13, 4].

Due to the difficulty of finding a convincing example, we have settled for a simple yet rather absurd one, as shown in Listing 8.27. The function takes two pointers to two (different) cells, and destroys the first.

The open statement is ambiguous: because of the wildcard, both Cell(p) and

```
void ambiguous(struct cell *p, struct cell *q)
  //@ requires Cell(p) &*& Cell(q);
  //@ ensures  Cell(q);
{
  //@ open Cell(_);
  free(p);
}
```

Listing 8.27: Ambiguous Match

Cell(q) match, and VeriFast is unsure which should be unfolded. In practice, VeriFast will pick an arbitrary chunk, open it, and proceed. The version[8] we used chose to unfold Cell(q), resulting in an error on the next line. In theory, VeriFast should backtrack and make a different choice: if it had chosen Cell(p), the function would have verified successfully.

Fortunately, such ambiguous matches do not appear often in practice. However, it will have a major impact on our formalisation of VeriFast in a later chapter.

8.3 Related Verification Tools

VeriFast is just one of many available verification tools. The quintessential example of another symbolic execution and separation logic based verifier is Smallfoot [12], which has many descendants such as SmallfootRG (adding rely-guarantee reasoning [69] for dealing with multithreaded code), Space Invader [11, 116], Abductor [28] and Heap-hop [109]. We will discuss these and others in more detail in Chapter 11, more specifically Sect. 11.3.

8.4 Conclusion

This chapter introduced the reader to VeriFast. Most importantly, the following points were discussed in detail and are relevant for the following chapters:

- Separation logic, on which VeriFast relies to describe program states, is detailed in Sect. 8.1. Its separating conjunction and frame rule considerably simplify dealing with shared state.

[8]VeriFast 12.5.23

- VeriFast verifies program by symbolically executing them. Executing a statement corresponds to transformating the current program state. Many statements also demand that the current program state satisfy certain conditions. Verification fails if those demands are not met.

- VeriFast requires every function or method to have a specification in the form of a precondition and postcondition. This leads to modular verification.

- During execution, the heap is represented abstractly by heap chunks. For example, a single allocated memory cell is represented by a $p \mapsto v$ chunk. User defined predicates make it possible to group chunks together into one single chunk, thereby making it possible to represent arbitrary data structures.

- As mentioned above, during symbolic execution, demands are often made of the program state. However, the same program state can be represented in many different ways, and VeriFast cannot always make out if it meets certain requirements. It is then necessary to rewrite the program state into an equivalent form so as to make it clear that it does indeed satisfy the conditions. This is done using the **`open`** and **`close`** ghost commands.

- Lemmas allow us to define more elaborate program state rewrites.

- To represent the contents of data structures at verification time, VeriFast provides inductive types. Fixpoint functions allow us to manipulate values of these types.

We are now ready for the next chapter, which presents a formalisation of a core part of VeriFast.

Chapter 9

Featherweight VeriFast

In this chapter we present Featherweight VeriFast, the theoretical core of VeriFast. While VeriFast works directly on C and Java code, we have introduced an intermediate verification language for Featherweight VeriFast in order to keep the formalisation as simple as possible. This intermediate verification language, which is named "Small Imperative Language", is the subject of Sect. 9.1 in which we define its syntax and informally describe its semantics.

Following this, we discuss the Result Algebra (Sect. 9.3), operators (Sect. 9.4) and basic operators (Sect. 9.5), which together form a semantic framework for the formalisation of the Small Imperative Language's semantics (Sect. 9.8, Sect. 9.9 and Sect. 9.10). A more detailed overview is given in Sect. 9.2, at which time the reader will have been provided with more context and our design decisions will make more sense.

Featherweight VeriFast has been fully formalised in Coq and can be extracted to O'Caml, meaning that we have a fully functional usable verifier. However, not all proofs have been machine checked. Some Coq extracts will appear in this chapter, but understanding them is not required. We refer the interested reader unfamiliar with Coq to Appendix C.

9.1 Small Imperative Language

While VeriFast directly supports both C and Java programming languages, our formalisation of it, Featherweight VeriFast, operates on a core subset of these languages, which we'll refer to as the "Small Imperative Language" (SIL). In this section, we only formally define SIL's syntax. Its formal semantics will be the subject of later sections (Sect. 9.8, Sect. 9.9 and Sect. 9.10).

Our verification algorithm will operate on SIL programs, but verifying SIL programs on its own is a purely academic endeavour. It only becomes meaningful if the results we get can somehow be applied on programs written in other programming languages.

To achieve this, we use SIL as an intermediate verification language (comparable with BoogiePL$^\flat$, see Chapter 3). For example, a C program will be translated into SIL, the resulting code will be verified, and the result of this verification should mean something about the original C program. The translation need not be complete: only the information required for verification needs to be preserved. However, SIL should still be a "full-featured" verification language in the sense that it needs to provide the necessary facilities to keep track of all information that *does* matter to verification. In short, this has led us to design SIL as a minimalistic imperative language with manual memory management.

Definition 9.1.1 (`Expression.t`)**.** *An* expression *is defined as*

$$\mathit{Expr} ::= \mathit{NatLiteral} \mid \mathit{Id} \mid \mathit{Expr} +_{\mathrm{e}} \mathit{Expr} \mid \mathit{Expr} -_{\mathrm{e}} \mathit{Expr} \mid \mathit{Expr} \times_{\mathrm{e}} \mathit{Expr}$$

Definition 9.1.2 (`BooleanExpression.t`)**.** *A* boolean expression *is defined as*

$$\begin{aligned}\mathit{BExpr} ::= {} & \mathbf{true}_{\mathrm{b}} \mid \mathit{Expr} =_{\mathrm{b}} \mathit{Expr} \mid \mathit{Expr} <_{\mathrm{b}} \mathit{Expr} \\ & \mid \mathit{Expr} \leq_{\mathrm{b}} \mathit{Expr} \mid \mathit{BExpr} \wedge_{\mathrm{b}} \mathit{BExpr} \mid \neg_{\mathrm{b}}\mathit{BExpr}\end{aligned}$$

The subscripts e and b for operators in expressions and boolean expressions, respectively, make explicit that we are dealing with syntactic entities, not mathematical expressions. This distinction will be especially important later on when we are dealing with mathematical expressions $a + b$, expressions $a +_{\mathrm{e}} b$ and terms $a +_{\mathrm{t}} b$.

Definition 9.1.3 (`SIL.command`)**.** *A* command *is defined as*

$$\begin{array}{lcll}
\mathit{Command} & ::= & \mathbf{skip} & \textit{no-op} \\
& \mid & \mathit{Id} := \mathit{Expr} & \textit{assignment} \\
& \mid & \mathit{Id} := [\mathit{Expr}] & \textit{reading from heap} \\
& \mid & [\mathit{Expr}] := \mathit{Expr} & \textit{writing to heap} \\
& \mid & \mathit{Command}; \mathit{Command} & \textit{sequencing} \\
& \mid & \mathit{RoutineName}(\mathit{Id}) & \textit{routine invocation} \\
& \mid & \mathbf{if}\ \mathit{BExpr}\ \mathbf{then}\ \mathit{Command}\ \mathbf{else}\ \mathit{Command} & \textit{conditional} \\
& \mid & x := \mathbf{malloc}(\mathit{NatLiteral}) & \textit{memory allocation} \\
& \mid & \mathbf{free}(\mathit{Expr}) & \textit{memory deallocation}
\end{array}$$

Definition 9.1.4 (`SIL.routine_definition`). *A* routine *has one parameter and a body.*

$$\textit{RoutineDefinition} ::= \textbf{routine}\ \textit{RoutineName}(\textit{Id}) = \textit{Command}$$

Definition 9.1.5 (`SIL.program`). *A SIL* program *is a set of routines, one of which is the main routine and is named* main. *We require that every routine call refers to a routine defined in the program.*

The semantics of a SIL program hides no surprises. Noteworthy is the behaviour of **malloc**: as in C, it returns a pointer to a newly allocated memory block, which can be located anywhere in memory (as long as it doesn't overlap with previously allocated memory). No guarantees are made about the memory block's initial contents: these can be arbitrary values.

The heap's address space coincides with $\mathbb{N}$, i.e. it is infinite in size and any natural number is a valid pointer. Likewise, values range over $\mathbb{N}$: a single memory cell can contain an arbitrarily large natural number.

An important aspect of programs is the fact that they generally interact with the external world: user input, files, network packets, etc. are received and reacted upon in a variety of ways. For the purposes of verification, it is important to be able to model this outside source of "arbitrary" data. Instead of extending the language with an extra command (comparable to a **havoc** command as defined in Chapter 4), it is possible to just rely on **malloc** to supply us with such nondeterminism. For example, reading a unit of data from a file is equivalent (for the purposes of verification at least) with allocating a new cell, storing its value v, deallocating the cell, and returning v.

A SIL program can fail in a number of ways:

- Reads ($\textit{Id}$:= [$\textit{Expr}$]) and writes ([$\textit{Expr}$] := $\textit{Expr}$) are only allowed on allocated memory cells, i.e. the relevant memory cell must be part of a block return by a **malloc** command and must not have been deallocated (**free**) in the meantime.
- One can only free previously allocated memory.

Thus, SIL can be compared to a "checked C": memory management is in hands of the programmer and pointer arithmetic is allowed, but derefences of wild or dangling pointers, out of bounds accesses, etc. are all caught. Verification then entails making sure that a given program does not encounter any of these failures during execution.

SIL misses some obvious features when compared to other programming languages. However, it is possible to simulate them:

- If multiple arguments need to be passed to a routine, it is possible to do so using a temporary memory block. For example, a routine call $r(x, y, z)$ can be encoded as follows:

  $$\begin{array}{l}\text{temp} := \textbf{malloc}(3);\\ [\text{temp}] := x;\ [\text{temp} +_{\text{e}} 1] := y;\ [\text{temp} +_{\text{e}} 2] := z;\\ r(\text{temp});\\ \textbf{free}(\text{temp})\end{array}$$

- Return values can be simulated using output parameters: we pass along an extra argument pointing to a memory cell to which the routine can write its return value. For example, using the syntactic extension introduced in the previous bullet point, we can encode $rv := r(x, y, z)$ as follows:

  $$\begin{array}{l}\text{temp} := \textbf{malloc}(1);\\ r(x, y, z, \text{temp});\\ rv := [\text{temp}];\\ \textbf{free}(\text{temp})\end{array}$$

- Loops can be replaced by routine calls. Given a loop

  $$pre;\ \textbf{while}\ b\ \textbf{do}\ c;\ post$$

  can be translated to SIL as follows:

  $$\begin{array}{l}pre;\\ \text{temp} := \textbf{malloc}(\#locals);\\ [\text{temp}] := local_1;\ [\text{temp} +_{\text{e}} 1] := local_2;\ \ldots;\ [\text{temp} +_{\text{e}} n-1] := local_n;\\ \text{aux}(\text{temp});\\ local_1 := [\text{temp}];\ local_2 := [\text{temp} + 1];\ \ldots;\ local_n := [\text{temp} +_{\text{e}} n-1];\\ \textbf{free}(\text{temp});\\ post\end{array}$$

  with as auxiliary routine definition

  $$\begin{array}{l}\textbf{routine}\ \text{aux}(\text{p}) =\\ \quad \textbf{if}\ b\ \textbf{then}\ (c';\ \text{aux}(\text{p}))\ \textbf{else skip}\end{array}$$

  where c' is a modified version of the loop body c where references to local variables are replaced by accesses to the heap through p.

In all these examples, temp and aux must of course be chosen fresh with respect to the other locals or routine names, respectively.

9.2 Overview

Our goal is to define a method to verify SIL programs and to prove this method sound. We first define a straightforward, intuitive way to verify SIL programs, which consists of just executing the program and see if failure is encountered along the way. We'll refer to this as the *concrete execution*. Its conceptual simplicity comes at a price: it is not computable, i.e. there is no upper bound on the time verification takes for nontrivial programs. This is due to the nondeterminism of **malloc**, which, as explained before, makes guarantees neither about where the newly allocated block is located in memory nor about its initial contents. To deal with this unknown factor, the execution forks into multiple paths, one for each "choice". In other words, execution is *nondeterministic*. Each such execution path[1] must be verified separately, none of them being allowed to fail. We will refer to this kind of choice as *demonic choice*. Since memory space is unbounded and values can be arbitrarily large natural numbers, there is an infinite number of such execution paths, which makes the concrete execution uncomputable.

In response to this, we adapt the concrete execution in order to make it usable in practice. This ultimately leads to the *symbolic execution*, which, while computable, is far more complex, making it harder to trust as a verification tool. Thus, it is necessary to show that the symbolic execution yields the same results as the concrete execution, i.e. that the symbolic execution is *sound* for the purposes of verification.

An important aspect of this symbolic execution is the fact that it introduces a new kind of nondeterminism: it is possible that symbolic execution encounters an ambiguous situation where it cannot know beforehand which choice to make (we hinted at this in Sect. 8.2.5). Again, execution will fork in multiple paths, but contrary to the previously discussed demonic choice, only one path needs to succeed. We call this an *angelic choice*. In short, angelic choice only requires that there exists *some* path leading to success, while demonic choice demands that *all* paths lead to success.

The transition from concrete to symbolic execution is a large one. To keep things manageable, we haven chosen to introduce an intermediate step, namely the *semiconcrete execution*.

To simplify the task of formalising these three executions, we define a shared semantic framework, as shown in Fig. 9.1. The *result algebra* forms a first

[1] A more intuitive interpretation: imagine that the **malloc** command is used to simulate user input, then the program must not fail for whatever input the user decides to enter. One branch must be created for each possible user input.

Concrete Execution (Sect. 9.8)	Semiconcrete Execution (Sect. 9.9)	Symbolic Execution (Sect. 9.10)
Basic Operators (Sect. 9.5)		
Operators (Sect. 9.4)		
Result Algebra (Sect. 9.3)		

Figure 9.1: Overview of the Different Abstraction Layers

abstraction layer. It allows us to express and compare the different executions' results (dealing with such complexities as demonic and angelic choice), thereby making it possible to relate the different executions with each other. On top of the result algebra we define *operators*, which can be seen as the building blocks for a monadic domain-specific language such as pioneered by Moggi [89] in the early nineties. Lastly, the three executions are defined in terms of operators.

9.3 Result Algebra

In this section, we introduce an algebraic structure named the *result algebra*. As explained in the previous section, executions exhibit two kinds of nondeterminism (demonic and angelic), and we need some way to model this information.

To provide better insight in the problem we attempt to solve with the result algebra, we start with a few examples in Sect. 9.3.1. Next, Sect. 9.3.2 formally defines the result algebra. To finish, a small selection of properties are stated and proved in Sect. 9.3.3.

9.3.1 Examples

In this section, we introduce the result algebra's central concepts through a series of examples.

Example 9.3.1. *This example introduces the concept of a "single result". We consider a language consisting of simple mathematical expressions:*

$$e ::= n \mid e + e \mid e \times e$$

Evaluation is perfectly deterministic: every expression evaluates to exactly one value. Let us denote this single value with $[\![x]\!]$, *i.e.*

$$\text{evaluate}(3 + 5 \times 8) = [\![43]\!]$$

Example 9.3.2. *We now introduce demonic choice. We generalise the language by replacing single numbers by sets of numbers:*

$$e ::= \{n^+\} \mid e + e \mid e \times e$$

Evaluating such an expression entails evaluating all expressions obtained by replacing a set by a value of that set. For example, $\{1,2\} \times \{3,4,5\}$ *yields* 3, 4, 5, 6, 8 *and* 10. *We model this result using the demonic choice operator:*

$$\text{evaluate}(\{1,2\} \times \{3,4,5\}) = [\![3]\!] \otimes [\![4]\!] \otimes [\![5]\!] \otimes [\![6]\!] \otimes [\![8]\!] \otimes [\![10]\!]$$

In our context, we view demonic choice as a "bad thing": we are dealing with a number of unknowns and would prefer precise answers. For example, knowing that a mathematical expression evaluates to "some number" (represented by $\bigotimes_{n \in \mathbb{N}} [\![n]\!]$) is less informative than to say it evaluates to 5 (i.e. $[\![5]\!]$).

Example 9.3.3. *In the previous examples, we have seen how to build results using the* $[\![-]\!]$ *and* $\otimes$ *operations, which in logical terms can be viewed as introduction rules. We need corresponding elimination rules, i.e. we need a way to extract information from a result.*

Say we want our mathematical expression to evaluate only to values within a certain range. The standard way to achieve this is by specifying a postcondition. We model a postcondition as a set of allowable outputs. We can then write

$$\text{postcondition} \models \text{result}$$

to indicate that result *satisfies* postcondition. *For example,*

$$\begin{array}{rcl} \{1\} & \models & [\![1]\!] \\ \{1,2\} & \models & [\![1]\!] \otimes [\![2]\!] \\ \{1\} & \not\models & [\![1]\!] \otimes [\![2]\!] \\ \{1,2\} & \models & [\![1]\!] \\ \mathbb{N} & \models & [\![1]\!] \\ X & \models & \bigotimes_{x \in X} [\![x]\!] \end{array}$$

Note how the postcondition is allowed to be an overapproximation.

Example 9.3.4. *We now demonstrate angelic choice. Let us again consider mathematical expressions with unknowns (as in Example 9.3.2). The expression also contains a number of parameters for which we are allowed to choose the values ourselves. We wish to know if we can pick values for these parameters such that the expression evaluates to values within a certain range. For this, we again extend our language:*

$$e ::= x \mid \textbf{pick}\ x \in \{n^+\}\ \textbf{in}\ e \mid \{n^+\} \mid e + e \mid e \times e$$

For example, the validity of

$$\mathbb{Z}^+ \models \text{evaluate}(\textbf{pick}\ x \in \{-1, 1\}\ \textbf{in}\ x \times \{3, -5\} + \{4\})$$

expresses that we can pick an $x \in \{-1, 1\}$ *such that both* $3x + 4$ *and* $-5x + 4$ *are positive. To represent the picking of x as a result, we use the* $\oplus$ *operator. The evaluation above thus yields the following result:*

$$\text{evaluate}(\textbf{pick}\ x \in \{-1, 1\}\ \textbf{in}\ x \times \{3, -5\} + \{4\}) = \bigoplus_{x \in \{-1,1\}} [\![3x + 4]\!] \oplus [\![-5x + 4]\!]$$

Admittedly, our description of the exact semantics of an expression are vague at best. It is clear however that an expression's result can become quite complex very quickly when combining the different choices. We defer formally defining the semantics until Sect. 9.5 (Fig. 9.3), at which point we will have the necessary tools to do this elegantly.

9.3.2 Operations and Axioms

We have discussed the result algebra's features through examples. This section defines the algebraic structure formally.

Definition 9.3.1 (result algebra)**.** *A* result algebra *over a set* $\mathcal{S}$ *defines a type of results, written* $\mathcal{S}^\star$*, and the following operations:*

$$[\![x]\!] \quad : \quad \mathcal{S} \to \mathcal{S}^\star \qquad \texttt{(RADefinitions.single)}$$

$$\bigoplus_{i \in I} R(i) \quad : \quad \forall\, I, R : I \to \mathcal{S}^\star.\ \mathcal{S}^\star \qquad \texttt{(RADefinitions.add)}$$

$$\bigotimes_{i \in I} R(i) \quad : \quad \forall\, I, R : I \to \mathcal{S}^\star.\ \mathcal{S}^\star \qquad \texttt{(RADefinitions.mul)}$$

$$\models \quad : \quad \mathcal{P}(\mathcal{S}) \to \mathcal{S}^\star \to \text{Prop} \qquad \texttt{(RADefinitions.models)}$$

```
Parameter RA : Type -> Type.
Variable  S  : Type.

Parameters
  (single  : S -> RA S)
  (models  : Ensemble S -> RA S -> Prop)
  (add     : forall {I : Type} (R : I -> RA S), RA S)
  (mul     : forall {I : Type} (R : I -> RA S), RA S)
  (top     : RA S)
  (bottom  : RA S)
  (implies : relation (RA S)).
```

Listing 9.1: Result Algebra Signature in Coq

The following axioms have to hold: for any type I and $S \in \mathcal{P}(\mathcal{S})$,

Ax-Single *(single_axiom)*

$$\forall\, x \in \mathcal{S}. \qquad S \models [\![x]\!] \iff x \in S$$

Ax-Angelic *(add_axiom)*

$$\forall\, R : I \to \mathcal{S}^\star. \qquad S \models \bigoplus_{i \in I} R(i) \iff \exists\, i \in I.\ S \models R(i)$$

Ax-Demonic *(mul_axiom)*

$$\forall\, R : I \to \mathcal{S}^\star. \qquad S \models \bigotimes_{i \in I} R(i) \iff \forall\, i \in I.\ S \models R(i)$$

Ax-Monotonicity *(monotonic_models_axiom)*

$$\forall\, R : \mathcal{S}^\star. \quad S' \subseteq S \quad \Rightarrow \quad S' \models R \quad \Rightarrow \quad S \models R$$

The Coq formalisations of these operations and axioms are shown[2] in Listing 9.1 and Listing 9.2, respectively.

The monotonicity axiom corresponds to the fact that “negative results” cannot be expressed: it is impossible to express that the postcondition must not contain certain outputs. In other words, results express lower bounds. This axiom was necessary to achieve the normalisation property (Lemma 9.3.10), which we will discuss shortly.

One could wonder why Ax-Single does not simply take the form $\{x\} \models [\![x]\!]$ and have Ax-Monotonicity take care of the generalisation to larger sets $S \supseteq \{x\}$:

[2]The reader unfamiliar with Coq might want to read Sect. C.6.1 and Sect. C.6.2 for clarification.

```
Axiom single_axiom :
  forall (s : S) (S : Ensemble S),
    models S single s <-> In s S.

Axiom top_axiom : forall (R : False -> RA S),
  equiv top (mul R).

Axiom bottom_axiom : forall (R : False -> RA S),
  equiv bottom (add R).

Axiom add_axiom :
  forall I (R : I -> RA S) (S : Ensemble S),
    models S (add R) <-> exists i, models S (R i).

Axiom mul_axiom :
  forall I (R : I -> RA S) (S : Ensemble S),
    models S (mul R) <-> forall i, models S (R i).

Axiom implies_axiom : forall (R R' : RA S),
  implies R R' <-> forall (S : Ensemble S),
                     models S R -> models S R'.

Axiom monotonic_models_axiom :
  forall (R : RA S) (S S' : Ensemble S),
    Included S' S -> models S' R -> models S R.
```

Listing 9.2: Result Algebra Axioms in Coq

now there seems to be overlap between the two axioms. However, the two formulations are not equivalent: given only $\{x\} \models [\![x]\!]$ instead of AX-SINGLE, it is impossible to prove that $S \models [\![x]\!] \Rightarrow x \in S$. In other words, the alternative formulation allows $S \models [\![x]\!]$ to be true by other means than $x \in S$, i.e. by the addition of other operations or axioms.

It would also have been possible to define a result as an inductive type; this would express that the *only* way to build a result is through one of the operators $[\![-]\!]$, $\oplus$ or $\otimes$ (the current treatment allows for extra operations to be added). The $\models$ relation could be defined as an inductive predicate or a fixpoint. This approach however would severly limit our freedom when defining models for the result algebra.

We define the following shorthand notation for binary choice:

Definition 9.3.2 (binary choice).

$$\begin{aligned} R_1 \oplus R_2 &\equiv \textstyle\bigoplus_{i\in\{1,2\}} R_i \\ R_1 \otimes R_2 &\equiv \textstyle\bigotimes_{i\in\{1,2\}} R_i \end{aligned}$$

where $R_1, R_2 : \mathcal{S}^\star$.

Example 9.3.5. *Let* $\mathcal{S} = \mathbb{N}$.

$$\begin{array}{lll} \{1\} & \models & [\![1]\!] \\ \{1\} & \models & [\![1]\!] \oplus [\![2]\!] \\ \{2\} & \models & [\![1]\!] \oplus [\![2]\!] \\ \{1\} & \not\models & [\![1]\!] \otimes [\![2]\!] \\ \{2\} & \not\models & [\![1]\!] \otimes [\![2]\!] \end{array} \qquad \begin{array}{lll} \{1,2,3\} & \models & [\![1]\!] \otimes \big(([\![2]\!] \otimes [\![3]\!]) \oplus ([\![4]\!] \otimes [\![5]\!])\big) \\ \{1,4,5\} & \models & [\![1]\!] \otimes \big(([\![2]\!] \otimes [\![3]\!]) \oplus ([\![4]\!] \otimes [\![5]\!])\big) \\ \{1,2,4\} & \not\models & [\![1]\!] \otimes \big(([\![2]\!] \otimes [\![3]\!]) \oplus ([\![4]\!] \otimes [\![5]\!])\big) \end{array}$$

We need a means to compare results. For this, we define an order relation on results. This relation is central in relating the different executions with each other.

Definition 9.3.3 (`RADefinitions.implies`, `RAAxioms.implies_axiom`). *A result R implies another result R′ if all postconditions satisfied by R are also satisfied by R′.*

$$R \Rrightarrow R' \quad\equiv\quad \forall\, S \in \mathcal{P}(\mathcal{S}).\; S \models R \;\Rightarrow\; S \models R'$$

where $R, R' : \mathcal{S}^\star$.

Lemma 9.3.1 (`RATheorems.implies_reflexivity`, `RATheorems.implies_transitivity`). $\mathcal{S}^\star$ *equipped with* $\Rrightarrow$ *is a partially preordered set.*

$$\begin{array}{lr} R \Rrightarrow R & \textit{reflexivity} \\ R \Rrightarrow R' \;\Rightarrow\; R' \Rrightarrow R'' \;\Rightarrow\; R \Rrightarrow R'' & \textit{transitivity} \end{array}$$

where $R, R', R'' : \mathcal{S}^\star$.

Two results are equivalent if they satisfy the same postconditions.

Definition 9.3.4 (`RADefinitions.equiv`). *If two results imply each other, we say they are* equivalent.

$$R \Longleftrightarrow R' \quad\equiv\quad R \Rrightarrow R' \wedge R' \Rrightarrow R$$

where $R, R' : \mathcal{S}^\star$.

Lemma 9.3.2 (`RATheorems.equiv_reflexivity`, `.equiv_symmetric`, `.equiv_transitivity`). *The* $\Longleftrightarrow$ *relation is an equivalence relation.*

$$\begin{array}{lr} R \Longleftrightarrow R & \textit{reflexivity} \\ R \Longleftrightarrow R' \;\Rightarrow\; R' \Longleftrightarrow R & \textit{symmetry} \\ R \Longleftrightarrow R' \;\Rightarrow\; R' \Longleftrightarrow R'' \;\Rightarrow\; R \Longleftrightarrow R'' & \textit{transitivity} \end{array}$$

where $R, R', R'' : \mathcal{S}^\star$.

We also define pointwise variants of the implication and equivalence relations:

Definition 9.3.5 (`RATheorems.f_implies`). *Given functions $R, R' : I \to \mathcal{S}^\star$, we define* pointwise implication *as*

$$R \mathrel{\dot{\Rightarrow}} R' \quad \equiv \quad \forall\, i : I.\; R\, i \Rightarrow R'\, i$$

Lemma 9.3.3 (`RATheorems.f_implies_reflexivity`, `.f_implies_transitivity`). *The $\dot{\Rightarrow}$ relation is a partial preorder. For any $R, R', R'' : I \to \mathcal{S}^\star$,*

$$\begin{array}{lr} R \mathrel{\dot{\Rightarrow}} R & \textit{reflexivity} \\ R \mathrel{\dot{\Rightarrow}} R' \;\Rightarrow\; R' \mathrel{\dot{\Rightarrow}} R'' \;\Rightarrow\; R \mathrel{\dot{\Rightarrow}} R'' & \textit{transitivity} \end{array}$$

Definition 9.3.6 (`RATheorems.f_equiv`). *Given functions $R, R' : I \to \mathcal{S}^\star$, we define* pointwise equivalence *as*

$$R \mathrel{\dot{\Longleftrightarrow}} R' \quad \equiv \quad \forall\, i : I.\; R\, i \Longleftrightarrow R'\, i$$

Lemma 9.3.4 (`RATheorems.f_equiv_reflexivity`, `RATheorems.f_equiv_symmetric`, `RATheorems.f_equiv_transitivity`). *The $\dot{\Longleftrightarrow}$ relation is an equivalence relation. For any $R, R', R'' : I \to \mathcal{S}^\star$,*

$$\begin{array}{lr} R \mathrel{\dot{\Longleftrightarrow}} R & \textit{reflexivity} \\ R \mathrel{\dot{\Longleftrightarrow}} R' \;\Rightarrow\; R' \mathrel{\dot{\Longleftrightarrow}} R & \textit{symmetry} \\ R \mathrel{\dot{\Longleftrightarrow}} R' \;\Rightarrow\; R' \mathrel{\dot{\Longleftrightarrow}} R'' \;\Rightarrow\; R \mathrel{\dot{\Longleftrightarrow}} R'' & \textit{transitivity} \end{array}$$

Let us examine the effect of choices over the empty set, i.e. what do $\bigotimes_{i\in\emptyset} R(i)$ and $\bigoplus_{i\in\emptyset} R(i)$ express?

- $S \models \bigotimes_{i\in\emptyset} R(i)$ requires that every $R(i)$ for $i \in \emptyset$ satisfies the postcondition S. This is trivially true. We denote this special result with $\top$ ("top").
- $S \models \bigoplus_{i\in\emptyset} R(i)$ demands that there exists some $i \in \emptyset$ such that $R(i)$ satisfies the postcondition S, which is impossible. We denote this with $\bot$ ("bottom").

Definition 9.3.7 (`RADefinitions.top`, `RAAxioms.top_axiom`). *$\top$ is defined as the demonic choice over the empty set.*

$$\top \equiv \bigotimes_{i\in\emptyset} R\, i$$

where R is the function with type signature $\emptyset \to \mathcal{S}^\star$.

Definition 9.3.8 (`RADefinitions.bottom`, `RAAxioms.bottom_axiom`). *$\bot$ is defined as the angelic choice over the empty set.*

$$\bot \equiv \bigoplus_{i \in \emptyset} R\ i$$

where R is the function with type signature $\emptyset \rightarrow \mathcal{S}^\star$.

Top and bottom are closely related to true and false, respectively, as the following lemmas show.

Lemma 9.3.5 (`RATheorems.only_bottom_without_models`). *Only bottom satisfies no postconditions.*

$$(\forall\ S : \mathcal{P}(\mathcal{S}).\ S \not\models R) \quad \Longleftrightarrow \quad R \Longleftrightarrow \bot$$

Lemma 9.3.6 (`RATheorems.bottom_implies_all`). *Every result is implied by bottom.*

$$\forall\ R : \mathcal{S}^\star.\ \bot \Rrightarrow R$$

Lemma 9.3.7 (`RATheorems.top_models_all`). *Only top satisfies all postconditions.*

$$(\forall\ S : \mathcal{P}(\mathcal{S}).\ S \models R) \quad \Longleftrightarrow \quad R \Longleftrightarrow \top$$

Lemma 9.3.8 (`RATheorems.top_model`). *Every result implies top.*

$$\forall\ R : \mathcal{S}^\star.\ R \Rrightarrow \top$$

Lemma 9.3.9 (`RATheorems.only_bottom_implies_bottom`). *Only bottom implies bottom.*

$$\forall\ R : \mathcal{S}^\star.\ R \Rrightarrow \bot \quad \Longleftrightarrow \quad R \Longleftrightarrow \bot$$

In the context of executions, top and bottom have interesting interpretations:

- Bottom corresponds to failing execution, as it will never satisfy any given postcondition.
- Top corresponds to nonterminating execution.

Thus, failure and nontermination can elegantly be represented using results: no special extra cases are necessary.

9.3.3 Lemmas

In this section we present a selection of lemmas (more have been proven in Coq). The normalisation property (Lemma 9.3.10) stands out and forms the basis of many proofs. It states that every result has a canonical form, i.e. every result can be written as an angelic choice of demonic choices of single results. This spares us the need for an induction principle for results.

We use the following shorthand notation and similar variations on it in the future:

$$\bigoplus_{S \models R} R'(S) \equiv \bigoplus_{S \in \{ S' \mid S' \models R \}} R'(S)$$

Lemma 9.3.10 (`RATheorems.normalization`)**.** *A result is always equivalent with the following normalised result:*

$$R \Longleftrightarrow \bigoplus_{S \models R} \bigotimes_{\sigma \in S} [\![\sigma]\!]$$

Proof. By Def. 9.3.3 and Def. 9.3.4, the proof goal is equivalent with

$$\forall\, S.\ S \models R \iff S \models \bigoplus_{S \models R} \bigotimes_{\sigma \in S} [\![\sigma]\!]$$

We take an arbitrary S. Applying Ax-Single, Ax-Demonic and Ax-Angelic (Def. 9.3.1), we get

$$S \models \bigoplus_{S \models R} \bigotimes_{\sigma \in S} [\![\sigma]\!] \iff \exists\, S' \models R.\ \forall\, \sigma \in S'.\ \sigma \in S$$

We split the proof goal into two parts:

- The $\Rightarrow$ part has proof goal

  $$S \models R \Rightarrow \exists\, S' \models R.\ \forall\, \sigma \in S'.\ \sigma \in S$$

  We take $S' = S$, the rest is trivially true.

- The $\Leftarrow$ part has proof state

  $$\begin{array}{rl} : & S' \models R \\ H_1 : & \forall\, \sigma \in S'.\ \sigma \in S \\ \hline & S \models R \end{array}$$

  From H_1 we know that $S' \subseteq S$. Combining this with Ax-Monotonicity gives us the proof goal.

□

The following lemma describes how top and bottom interact with choices.

Lemma 9.3.11 (`RATheorems.add_top`, `.add_bottom`, `.mul_bottom`, `.mul_top`). *Given a function* $R : I \to \mathcal{S}^\star$,

$$\begin{array}{ccc}
(\exists\, i : I,\ R\,i \Longleftrightarrow \top) & \Rightarrow & \bigoplus_{i\in I} R\,i \Longleftrightarrow \top \\
(\forall\, i : I,\ R\,i \Longleftrightarrow \bot) & \Longleftrightarrow & \bigoplus_{i\in I} R\,i \Longleftrightarrow \bot \\
(\exists\, i : I,\ R\,i \Longleftrightarrow \bot) & \Rightarrow & \bigotimes_{i\in I} R\,i \Longleftrightarrow \bot \\
(\forall\, i : I,\ R\,i \Longleftrightarrow \top) & \Longleftrightarrow & \bigotimes_{i\in I} R\,i \Longleftrightarrow \top
\end{array}$$

Proof. Using Def. 9.3.4 and Def. 9.3.3 we can reformulate the proof goals using the following equivalence:

$$X \Longleftrightarrow Y \qquad \Longleftrightarrow \qquad \forall\, S \in \mathcal{P}(\mathcal{S}).\ S \models X \iff S \models Y$$

We prove each proposition in turn:

- Proof goal $(\exists\, i : I,\ R\,i \Longleftrightarrow \top) \Rightarrow \left(\forall\, S \in \mathcal{P}(\mathcal{S}).\ S \models \bigoplus_{i\in I} R\,i \iff S \models \top\right)$. The $\Rightarrow$ part follows from Lemma 9.3.7. The $\Leftarrow$ part follows from Ax-Angelic and the premise.
- Proof goal $(\forall\, i : I,\ R\,i \Longleftrightarrow \bot) \iff \bigoplus_{i\in I} R\,i \Longleftrightarrow \bot$. We split the $\iff$ in two:
  - Goal: $(\forall\, i : I,\ R\,i \Longleftrightarrow \bot) \Rightarrow \left(\forall\, S \in \mathcal{P}(\mathcal{S}).\ S \models \bigoplus_{i\in I} R\,i \iff S \models \bot\right)$. The $\Rightarrow$ part: from Ax-Angelic we know that $\exists\, i \in I.\ S \models R\,i$. We take this i. Since $R\,i \Longleftrightarrow \bot$ we have $S \models \bot$, which is impossible according to Lemma 9.3.5. The $\Leftarrow$ part follows directly from Lemma 9.3.5.
  - Proof state:

$$\begin{array}{ll}
H_1 : & \forall\, S_x \in \mathcal{P}(\mathcal{S}).\ (\exists\, i_x \in I.\ S_x \models R\,i_x) \iff S_x \models \bot \\
 & S \in \mathcal{P}(\mathcal{S}) \\
 & i : I \\
\hline
 & S \models R\,i \iff S \models \bot
\end{array}$$

    Proving the $\Rightarrow$ part: if we take $S_x = S$ and $i_x = i$ in H_1, we get $S \models \bot$, which is absurd (Lemma 9.3.5). Proving the $\Leftarrow$ part: $S \models \bot$ is absurd (Lemma 9.3.5).

- Proof goal $(\exists\, i : I,\ R\, i \Longleftrightarrow \bot) \Rightarrow \big(\forall\, S \in \mathcal{P}(\mathcal{S}).\ S \models \bigotimes_{i \in I} R\, i \iff S \models \bot\big)$. The $\Rightarrow$ part: from Ax-Demonic follows $\forall\, i \in I.\ S \models R\, i$. Combined with the antecedent, we get $S \models \bot$, which is impossible (Lemma 9.3.5). The $\Leftarrow$ part follows directly from Lemma 9.3.5.
- Proof goal $(\forall\, i : I,\ R\, i \Longleftrightarrow \top) \iff \bigotimes_{i \in I} R\, i \Longleftrightarrow \top$. We split the $\iff$ in two:
  - Goal: $(\forall\, i : I,\ R\, i \Longleftrightarrow \top) \Rightarrow \forall\, S \in \mathcal{P}(\mathcal{S}).\ S \models \bigotimes_{i \in I} R\, i \iff S \models \top$. The $\Rightarrow$ part follows from Lemma 9.3.7. The $\Leftarrow$ part follows from Ax-Demonic and the premise.
  - Proof state:

$$\begin{array}{ll} H_1 : & \forall\, S_x \in \mathcal{P}(\mathcal{S}).\ S \models \bigotimes_{i \in I} R\, i \iff S \models \top \\ & S \in \mathcal{P}(\mathcal{S}) \\ & i : I \\ \hline & S \models R\, i \iff S \models \top \end{array}$$

    The $\Rightarrow$ part follows from Lemma 9.3.7. The $\Leftarrow$ part: from H_1 with $S_x = S$ we get $S \models \bigotimes_{i \in I} R\, i$. Ax-Demonic finishes the proof.

□

Lemma 9.3.12 (`RATheorems.only_empty_set_models_top`). *Only top satisfies the empty postcondition* $\emptyset$*.*

$$\emptyset \models R \iff R \Longleftrightarrow \top$$

Proof. The $\Rightarrow$ part: Ax-Monotonicity and $\emptyset \models R$ give $\forall\, S \in \mathcal{P}(\mathcal{S}).\ S \models R$. According to Lemma 9.3.7, R must be equivalent with $\top$. The $\Leftarrow$ part: according to Lemma 9.3.7, top satisfies all postconditions, hence $\emptyset$ too. □

We can categorise results based on which nondeterminism they represent. This categorisation will prove important in Sect. 9.5.

Definition 9.3.9 (`RATheorems.demonic`). *A result is* demonic *if it only involves demonic choice.*

$$\textbf{demonic}(R) \equiv \exists\, \mathcal{K}.\ R \Longleftrightarrow \bigotimes_{r \in \mathcal{K}} [\![r]\!]$$

Lemma 9.3.13 (`RATheorems.demonic_equiv_iff_same_charset`). *Given $\mathcal{K}$ and $\mathcal{K}'$, being the characteristic sets of demonic results R and R' respectively, then*

$$R \Longleftrightarrow R' \iff \mathcal{K} = \mathcal{K}'$$

Lemma 9.3.14 (RATheorems.unique_demonic_K). *The characteristic set of a demonic result R is unique. We denote it $|R|_{\mathrm{d}}$.*

Definition 9.3.10 (RATheorems.angelic). *A result is* angelic *if it only involves angelic choice.*

$$\mathbf{angelic}(R) \equiv \exists\, \mathcal{K}.\, R \Longleftrightarrow \bigoplus_{r \in \mathcal{K}} [\![r]\!]$$

The set $\mathcal{K}$ is called a characteristic set *of R.*

Lemma 9.3.15 (RATheorems.angelic_equiv_iff_same_charset). *Given $\mathcal{K}$ and $\mathcal{K}'$, being the characteristic sets of angelic results R and R' respectively, then*

$$R \Longleftrightarrow R' \quad \Longleftrightarrow \quad \mathcal{K} = \mathcal{K}'$$

Lemma 9.3.16 (RATheorems.unique_angelic_K). *The characteristic set of an angelic result R is unique. We denote it $|R|_{\mathrm{a}}$.*

Definition 9.3.11 (RATheorems.deterministic). *A result is* deterministic *if it involves neither angelic nor demonic choice.*

$$\mathbf{deterministic}(R) \equiv \exists\, \mathcal{K}.\, R \Longleftrightarrow [\![\mathcal{K}]\!]$$

Lemma 9.3.17 (RATheorems.deterministic_equiv_iff_same_charset). *Given $\mathcal{K}$ and $\mathcal{K}'$, being the characteristic values of deterministic results R and R' respectively, then*

$$R \Longleftrightarrow R' \quad \Longleftrightarrow \quad \mathcal{K} = \mathcal{K}'$$

Lemma 9.3.18 (RATheorems.unique_deterministic_K). *The characteristic value of a deterministic result R is unique. We denote it $|R|_{\mathrm{det}}$.*

Lemma 9.3.19 (RATheorems.deterministic_implies_angelic, .deterministic_-implies_demonic, .demonic_angelic_implies_deterministic). *A result is deterministic iff it is both angelic and demonic.*

$$\mathbf{deterministic}(R) \qquad \Longleftrightarrow \qquad \mathbf{angelic}(R) \wedge \mathbf{demonic}(R)$$

Lemma 9.3.20 (RAOperatorTheorems.swap_add, RAOperatorTheorems.swap_mul). *Nondependent choices commute.*

$$\bigotimes\nolimits_{i \in I} \bigotimes\nolimits_{j \in J} R(i,j) \quad \Longleftrightarrow \quad \bigotimes\nolimits_{j \in J} \bigotimes\nolimits_{i \in I} R(i,j)$$

$$\bigoplus\nolimits_{i \in I} \bigoplus\nolimits_{j \in J} R(i,j) \quad \Longleftrightarrow \quad \bigoplus\nolimits_{j \in J} \bigoplus\nolimits_{i \in I} R(i,j)$$

9.4 Operators

In the previous section, we have introduced the result algebra, which allows us to express results involving demonic and angelic choice. While it is possible to formalise the semantics of the three executions (as explained in Sect. 9.2) by directly expressing them in terms of results, readability would suffer.

Our intention is to obtain an executable symbolic execution. Therefore, we wish the formalisation of the semantics to take the form of an executable program. For this purpose, we introduce *operators*, i.e. composable functions which abstract away details such as state handling and nondeterminism. While the concrete and semiconcrete execution are inherently uncomputable, we define their semantics using the same framework, which simplifies the task of relating the different executions with each other.

Operators are monadic functions. Readers unfamiliar with the concept of monads may find this section daunting; we refer them to Appendix B for a quick introduction to monads. The monad we use in this section most resembles a combination of the State and the List monad.

Definition 9.4.1 (`RAOperators.primitive_operator`). *A* primitive operator from $\mathcal{S}_1$ to $\mathcal{S}_2$ *is defined as*

$$\mathit{Operator}^{\mathcal{S}_1 \to \mathcal{S}_2} \equiv \mathcal{S}_1 \to \mathcal{S}_2^{\star}$$

Note that we have deviated from the typical approach such as Haskell's:

- There is no explicit return value. We can easily solve this problem by incorporating it into the output type, i.e. by taking $\mathcal{S}_2 = A \times \mathcal{S}_2'$.
- The state type varies: a more standard type signature for operators would be $\mathcal{S} \to \mathcal{S}^{\star}$. We allow the input and output to differ for two reasons. Firstly, it allows us to include the result type into the output state (as explained above). Secondly, while the concrete, semiconcrete and symbolic executions operate on concrete, semiconcrete and symbolic states, respectively, we also require the possibility of defining cross-execution operators, e.g. which take a symbolic state as input yet return a concrete state as output.

Primitive operators can be composed using the binding operation, yielding a new primitive operator[3]. This corresponds to sequencing: the output of the

[3]The name "primitive operator" might be a bit confusing: a primitive operator is a function with a specific type (Def. 9.4.1); binding two primitive operators yields a new function with a type that satisfies that of primitive operators, hence it is also a primitive operator.

first operator is applied to the second while taking failure, nontermination and the different choices into account.

We define binding operation in terms of a lift operation. We do not give a precise definition for this lift operation, but instead only require that it obeys certain axioms. This gives us the freedom to give specialised definitions for different models of the result algebra.

Definition 9.4.2 (`RAOperators.lift`). *The* lift *operation, denoted* $-^{\Uparrow}$*, has the following type signature:*

$$-^{\Uparrow} : Operator^{\mathcal{S}_1 \rightarrow \mathcal{S}_2} \rightarrow \mathcal{S}_1{}^{\star} \rightarrow \mathcal{S}_2{}^{\star}$$

It must satisfy the following axioms:

Ax-LiftSingle *(`RAOperators.lift_single_axiom`)*

$$op^{\Uparrow}([\![x]\!]) \iff op(x)$$

Ax-LiftAngelic *(`RAOperators.lift_add_axiom`)*

$$op^{\Uparrow}\left(\bigoplus_{i\in I} R(i)\right) \iff \bigoplus_{i\in I} op^{\Uparrow}(R(i))$$

Ax-LiftDemonic *(`RAOperators.lift_mul_axiom`)*

$$op^{\Uparrow}\left(\bigotimes_{i\in I} R(i)\right) \iff \bigotimes_{i\in I} op^{\Uparrow}(R(i))$$

Ax-LiftMonotonic *(`RAOperators.monotonic_lift_axiom`)*

$$op \mathrel{\dot{\Rrightarrow}} op' \quad \Rightarrow \quad R \Rrightarrow R' \quad \Rightarrow \quad op^{\Uparrow}(R) \Rrightarrow (op')^{\Uparrow}(R')$$

The first three axioms are straightforward. The reader might wonder why Ax-LiftMonotonic is required: it seems to follow from the other axioms, especially Ax-Monotonicity (p. 111). We examine this in the following lemma.

Lemma 9.4.1 (`FakeModel.monotonic_lift_axiom_fails`). *The monotonicity axiom* Ax-LiftMonotonic *does* not *follow from the other axioms.*

Proof. Let us consider a weaker version of Ax-LiftMonotonic:

$$R \Rrightarrow R' \quad \Rightarrow \quad op^{\Uparrow}(R) \Rrightarrow op^{\Uparrow}(R') \tag{9.1}$$

i.e. we take $op = op'$. If this variant cannot be proven, it is clear that Ax-LiftMonotonic cannot be proven either. Let us try to prove that (9.1) follows from the other axioms. The proof state becomes

$$\frac{R \Rrightarrow R'}{op^{\Uparrow}(R) \Rrightarrow op^{\Uparrow}(R')}$$

We could use the normalisation lemma (Lemma 9.3.10 on page 116) to get a canonical form for R and R', but it will not be of much use: it would only show that R and R' are *equivalent* ($\Longleftrightarrow$) with their respective canonical form, not equal (=) to it. Thus, it is not possible to rewrite $op^{\Uparrow}(R)$ to $op^{\Uparrow}(\bigoplus_{S \models R} \bigotimes_{\sigma \in S} [\![\sigma]\!])$ in our proof goal, since it would require the following to hold:

$$R \Longleftrightarrow R' \quad \Rightarrow \quad op(R) \Longleftrightarrow op(R')$$

This is exactly what Ax-LiftMonotonic can provide us with, and is thus the reason for its existence.

Admittedly, this line of reasoning does not form a proof; it merely shows where a straightforward attempt to prove that (9.1) fails. To demonstrate that it is indeed impossible to prove it, we show we can construct a model which satisfies all axioms except Ax-LiftMonotonic. It is possible to define a model with two $[\![-]\!]$ constructors, let's call them red and blue. The $\models$ relation would behave identically for both, but the lift operation might be defined so that $op^{\Uparrow}([\![x]\!]_{\mathrm{r}}) = op(x)$ and $op^{\Uparrow}([\![x]\!]_{\mathrm{b}}) = \bot$. All axioms would be satisfied, except for Ax-LiftMonotonic. Details of this model can be found in Sect. D.17. □

We prove a few interesting properties of lifting which follow from the axioms.

Lemma 9.4.2 (`RAOperatorTheorems.lift_normalized`)**.**

$$op^{\Uparrow} \overset{\cdot}{\Longleftrightarrow} \lambda\ R.\ \bigoplus_{S \models R} \bigotimes_{\sigma \in S} op(\sigma)$$

Proof.

By Def. 9.3.6 and taking an arbitrary R, the proof goal becomes

$$op^{\Uparrow}(R) \Longleftrightarrow \bigoplus_{S \models R} \bigotimes_{\sigma \in S} op(\sigma)$$

Applying Lemma 9.3.10 on R in the lhs gives

$$op^{\Uparrow}\left(\bigoplus_{S \models R} \bigotimes_{\sigma \in S} [\![\sigma]\!]\right) \Longleftrightarrow \bigoplus_{S \models R} \bigotimes_{\sigma \in S} op(\sigma)$$

Using Ax-LiftAngelic, Ax-LiftDemonic and Ax-LiftSingle in turn, we get

$$\bigoplus_{S \models R} \bigotimes_{\sigma \in S} op(\sigma) \Longleftrightarrow \bigoplus_{S \models R} \bigotimes_{\sigma \in S} op(\sigma)$$

which is trivially true (reflexivity of $\Longleftrightarrow$). □

Lemma 9.4.3 (`RAOperatorTheorems.lift_top`, `.lift_bottom`). *Applying a lifted operator on top or bottom yields top or bottom, respectively.*

$$\begin{array}{rcl} op^{\Uparrow}(\top) & \Longleftrightarrow & \top \\ op^{\Uparrow}(\bot) & \Longleftrightarrow & \bot \end{array}$$

Proof.

$$\begin{array}{rcll} op^{\Uparrow}(\top) & \Longleftrightarrow & op^{\Uparrow}\left(\bigotimes_{i \in \emptyset} R(i)\right) & \text{by Def. 9.3.7} \\ & \Longleftrightarrow & \bigotimes_{i \in \emptyset} op^{\Uparrow}(R(i)) & \text{by Ax-LiftDemonic} \\ & \Longleftrightarrow & \top & \text{by Def. 9.3.7} \end{array}$$

Similarly for bottom:

$$\begin{array}{rcll} op^{\Uparrow}(\bot) & \Longleftrightarrow & op^{\Uparrow}\left(\bigoplus_{i \in \emptyset} R(i)\right) & \text{by Def. 9.3.8} \\ & \Longleftrightarrow & \bigoplus_{i \in \emptyset} op^{\Uparrow}(R(i)) & \text{by Ax-LiftAngelic} \\ & \Longleftrightarrow & \bot & \text{by Def. 9.3.8} \end{array}$$

□

We now define primitive operator binding in terms of operator lifting.

Definition 9.4.3 (`RAOperatorTheorems.bind`). Primitive operator binding, *denoted* $\ggg$*, composes primitive operators as follows:*

$$\begin{array}{cccc|l} - & \ggg & - & : & Operator^{\mathcal{S}_1 \to \mathcal{S}_2} \to Operator^{\mathcal{S}_2 \to \mathcal{S}_3} \to Operator^{\mathcal{S}_1 \to \mathcal{S}_3} \\ \hline op_1 & \ggg & op_2 & \overset{\cdot}{\Longleftrightarrow} & {op_2}^{\Uparrow} \circ op_1 \end{array}$$

Binding inherits important properties from operator lifting.

Lemma 9.4.4 (`RAOperatorTheorems.bind_top_left`, `.bind_bottom_left`). *Once an execution path encounters failure (bottom) or nontermination (top), it remains stuck there: no further computation has any effect.*

$$op(\sigma) \Longleftrightarrow \bot \quad \Rightarrow \quad (op \ggg op')(\sigma) \Longleftrightarrow \bot$$

$$op(\sigma) \Longleftrightarrow \top \quad \Rightarrow \quad (op \ggg op')(\sigma) \Longleftrightarrow \top$$

Proof. The $\bot$ case:

$$\begin{array}{rcll} (op \ggg op')(\sigma) & \Longleftrightarrow & (op')^{\Uparrow}(op(\sigma)) & \text{by Def. 9.4.3} \\ & \Longleftrightarrow & (op')^{\Uparrow}(\bot) & \text{premise} \\ & \Longleftrightarrow & \bot & \text{by Lemma 9.4.3} \end{array}$$

Idem for the $\top$ case. □

Lemma 9.4.5 (`RAOperatorTheorems.bind_failure_left`, `.bind_block_left`). *The binding of a failing or nonterminating operator with another operator has no effect.*

$$\begin{array}{rcl} (\lambda\ \sigma.\ \bot) \ggg op & \dot{\Longleftrightarrow} & \lambda\ \sigma.\ \bot \\ (\lambda\ \sigma.\ \top) \ggg op & \dot{\Longleftrightarrow} & \lambda\ \sigma.\ \top \end{array}$$

Proof. Follows from Lemma 9.4.4. □

Lemma 9.4.6 (`RAOperatorTheorems.bind_failure_right`, `.bind_block_right`). *Binding with a nonterminating or failing operator on the right yields nontermination or failure, respectively, on condition that the left operator does not fail or diverge first, respectively.*

$$\begin{array}{rcl} op(\sigma) \not\Longleftrightarrow \top & \Rightarrow & (op \ggg (\lambda\ \sigma.\ \bot))(\sigma) \Longleftrightarrow \bot \\ op(\sigma) \not\Longleftrightarrow \bot & \Rightarrow & (op \ggg (\lambda\ \sigma.\ \top))(\sigma) \Longleftrightarrow \top \end{array}$$

Proof. We focus on the first proposition. From Def. 9.4.3, we know

$$(op \ggg (\lambda\ \sigma.\ \bot))(\sigma) \Longleftrightarrow \bigoplus_{S \models op(\sigma)} \bigotimes_{\sigma' \in S} \bot$$

For this to be equivalent with $\bot$, Lemma 9.3.11 (second proposition) tells us that $\bigotimes_{\sigma' \in S} \bot$ must be equivalent with $\bot$ for any $S \models op(\sigma)$. Lemma 9.3.11 (third proposition) tells us that in order for this to be the case, there must exist a $\sigma' \in S$. Hence, we must prove S to be inhabited. If S were empty, then $\emptyset \models op(\sigma)$, and according to Lemma 9.3.12, this means $op(\sigma) \Longleftrightarrow \top$, which contradicts the premise.

We turn our attention to the second proposition. From Def. 9.4.3,

$$(op \ggg (\lambda\ \sigma.\ \top))(\sigma) \Longleftrightarrow \bigoplus_{S \models op(\sigma)} \bigotimes_{\sigma' \in S} \top$$

We need to prove this equivalent with $\top$. Lemma 9.3.11 (first proposition) tells us there needs to exist some $S \models op(\sigma)$ such that $\bigotimes_{\sigma' \in S} \top \Longleftrightarrow \top$. If no such S were to exist, Lemma 9.3.5 tells us that $op(\sigma) \Longleftrightarrow \bot$, which would contradict the premise. To show that $\bigotimes_{\sigma' \in S} \top \Longleftrightarrow \top$, we are required according to Lemma 9.3.11 (fourth proposition) to prove that $\top \Longleftrightarrow \top$ for any $\sigma' \in S$, which is trivially true. □

We now introduce *operators* which can return values. We also define a corresponding binding operation. For operators that do not yield any useful return values (but instead transforming the state, i.e. exhibit side effects) we make use of a unit data type.

Definition 9.4.4. *The unit set* unit *has as only element* $\square$.

Definition 9.4.5 (RAOperators.operator)**.** *An* operator from $\mathcal{S}_1$ to $\mathcal{S}_2$ with result *A is defined as*

$$Operator_A^{\mathcal{S}_1 \to \mathcal{S}_2} \equiv Operator^{\mathcal{S}_1 \to (A \times \mathcal{S}_2)^\star} \equiv \mathcal{S}_1 \to (A \times \mathcal{S}_2)^\star$$

Definition 9.4.6 (RAOperators.bind)**.** Operator binding $\ggg=$ *is defined as*

$$\begin{array}{lcl} - \ggg= - & : & Operator_A^{\mathcal{S}_1 \to \mathcal{S}_2} \to (A \to Operator_B^{\mathcal{S}_2 \to \mathcal{S}_3}) \to Operator_B^{\mathcal{S}_1 \to \mathcal{S}_3} \\ op \ggg= f & \Longleftrightarrow & op \ggg (\lambda\ (a, \sigma).\ f\ a\ \sigma) \end{array}$$

A bind operation which ignores return values is defined as

$$op_1 \gg op_2 \ \equiv\ op_1 \ggg= (\lambda\ x.\ op_2)$$

with operators returning $\square \in$ unit.

As with many languages which support monads, we define a special syntax which will benefit the readability of further formalisations.

Definition 9.4.7 (do notation)**.** *We define the* do notation, *both single line and multiline:*

$$\begin{array}{lcllcl} \textbf{do}\ \ x \leftarrow f; rest & \equiv & \textbf{do}\ \ x \leftarrow f & \equiv & f \ggg= \lambda\ x.\ rest \\ & & \quad\ rest & & \\ & & & & \\ \textbf{do}\ \ f; rest & \equiv & \textbf{do}\ \ f & \equiv & f \gg rest \\ & & \quad\ rest & & \\ & & & & \\ & & \textbf{do}\ \ f & \equiv & f \end{array}$$

For the sake of completeness, we prove that our monad satisfies the monad laws [113].

Theorem 9.4.1 (RAOperatorTheorems.monad_theorem_1, .monad_theorem_2, .monad_theorem_3)**.** *The type constructor*

$$T\ a = (A \times \mathcal{S})^\star$$

the binding $\gg=$ *and unit function*

$$\mathrm{ret} \equiv \lambda\ r.\ \lambda\ \sigma.\ [\![(r,\sigma)]\!]$$

form a valid Kleisli triple.

$$\begin{array}{rcl} \mathrm{ret}\ x \gg= f & \dot{\Longleftrightarrow} & f\ x \\ op \gg= \mathrm{ret} & \dot{\Longleftrightarrow} & op \\ (op \gg= f) \gg= g & \dot{\Longleftrightarrow} & op \gg= (\lambda\ x.\ f\ x \gg= g) \end{array}$$

9.5 Basic Operators

In this section, we introduce a last abstraction layer, designed to shield us from the details of the result algebra. We wish to provide all functionality (demonic and angelic choice, state, etc.) through a small set of *basic operators*[4]. Together they form a small domain specific language.

Definition 9.5.1 (basic operators)**.** *Figure 9.2 gives a list of all* basic operators.

We clarify the semantics of these basic operators. $\mathsf{pick}_\mathrm{d}(A)$ represents demonic choice over a set A. $\mathsf{pick}_\mathrm{d}(A) \gg= f$ succeeds iff $f(a)$ succeeds for every $a \in A$. Angelic choice is made available through $\mathsf{pick}_\mathrm{a}(A)$: $\mathsf{pick}_\mathrm{a}(A) \gg= f$ succeeds iff there is some $a \in A$ for which $f(a)$ succeeds.

Example 9.5.1. *Reading a single byte from a file can be modelled as follows:*

$$\begin{array}{rcl} \textsf{read-byte} & : & \mathit{Operator}_{\mathbb{Z}}^{\mathcal{S}\to\mathcal{S}} \\ \textsf{read-byte} & = & \mathsf{pick}_\mathrm{d}(\mathbb{Z}_{256}) \end{array}$$

To "execute" this program, we unfold its type signature (Def. 9.4.5 and Def. 9.4.1):

$$\textsf{read-byte} : \mathcal{S} \to (\mathbb{Z} \times \mathcal{S})^\star$$

Since we don't use state, we can choose $\mathcal{S}$ *arbitrarily, say* unit. *Let us verify this function: we expect* read-byte *to return some value in the* $[0,255]$ *range. Thus, the result of executing* read-byte *should satisfy the postcondition* $\{1,2,\ldots,255\}$. *However, the types don't match: the postcondition is a set of integers, while the result contains pairs of type* $\mathbb{Z} \times$ unit. *We can resolve this in two ways. Either we adapt the postcondition:*

$$\{(1,\square),(2,\square),\ldots,(255,\square)\} \models \textsf{read-byte}(\square)$$

[4]For the sake of clarity: "primitive operator" and "operator" define function type signatures. A basic operator is a function with that type, i.e. a basic operator is an instance of a (primitive) operator.

return	:	$A \to \mathit{Operator}_A^{\mathcal{S}\to\mathcal{S}}$	`yield`
return a	=	$\lambda\ \sigma.\ [\![(a,\sigma)]\!]$	
nop	:	$\mathit{Operator}_{\mathrm{unit}}^{\mathcal{S}\to\mathcal{S}}$	`nop`
nop	=	return $\square$	
pick$_{\mathrm{a}}$	:	$\mathcal{P}(A) \to \mathit{Operator}_A^{\mathcal{S}\to\mathcal{S}}$	`pick_angelically`
pick$_{\mathrm{a}}(A)$	=	$\lambda\ \sigma.\ \bigoplus_{a\in A} [\![(a,\sigma)]\!]$	
pick$_{\mathrm{d}}$	:	$\mathcal{P}(A) \to \mathit{Operator}_A^{\mathcal{S}\to\mathcal{S}}$	`pick_demonically`
pick$_{\mathrm{d}}(A)$	=	$\lambda\ \sigma.\ \bigotimes_{a\in A} [\![(a,\sigma)]\!]$	
current-state	:	$\mathit{Operator}_{\mathcal{S}}^{\mathcal{S}\to\mathcal{S}}$	`current_state`
current-state	=	$\lambda\ \sigma.\ [\![(\sigma,\sigma)]\!]$	
set-current-state	:	$\mathcal{S}_2 \to \mathit{Operator}_{\mathrm{unit}}^{\mathcal{S}_1\to\mathcal{S}_2}$	`set_current_state`
set-current-state σ'	=	$\lambda\ \sigma.\ [\![(\square,\sigma')]\!]$	
block	:	$\mathit{Operator}_A^{\mathcal{S}\to\mathcal{S}}$	`block`
block	=	$\lambda\ \sigma.\ \top$	
fail	:	$\mathit{Operator}_A^{\mathcal{S}\to\mathcal{S}}$	`failure`
fail	=	$\lambda\ \sigma.\ \bot$	
assert	:	$\mathrm{bool} \to \mathit{Operator}_{\mathrm{unit}}^{\mathcal{S}\to\mathcal{S}}$	`assert`
assert **true**	=	nop	
assert **false**	=	fail	
assume	:	$\mathrm{bool} \to \mathit{Operator}_{\mathrm{unit}}^{\mathcal{S}\to\mathcal{S}}$	`assume`
assume **true**	=	nop	
assume **false**	=	block	

Figure 9.2: Basic Operators (in Coq Module `RAOperators`)

or we can make use of a primitive operator which is able to break out of the $A \times \mathcal{S}$ *mold which (nonprimitive) operators require.*

$$\mathsf{unpack} = \lambda\ (n, s).\ [\![n]\!]$$

We can now check the postcondition as follows:

$$\{1, 2, \ldots, 255\} \models (\mathsf{read\text{-}byte} \ggg \mathsf{unpack})(\square)$$

block represents nontermination. Its name originates from its use: we rely on it to block further execution and hence prevent failure from occurring later on. Formally, this is expressed by the fact that $\mathsf{block} \gg\!= op \overset{\cdot}{\iff} \mathsf{block}$. It is seldom used directly; a more useful conditional block is available through assume.

The fail operator leads the current execution path to unrecoverable failure: $\mathsf{fail} \gg\!= op \overset{\cdot}{\iff} \mathsf{fail}$. This does not necessarily mean the entire execution must fail: a previously made angelic choice may provide alternative nonfailing paths. Like block, it has a conditional variant, namely assert.

Example 9.5.2. *This example discusses how we can implement integer factorisation. Abstractly, we wish to define a function with the following specifications:*

$$\text{factor} : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$$

$$\text{factor}(n) = (k, m) \iff k \geq 2 \land k \cdot m = n$$

The operator factor-alg(n) *is a possible implementation. If no pair* (k, m) *exists satisfying the contract (i.e. when* $n = 1$*), then the computation does not terminate.*

```
factor-alg(n)    =  do factor-aux(n, 2)

factor-aux(n, k) =  do if n mod k = 0
                       then return (k, n/k)
                       else factor-aux(n, k + 1)
```

We can verify specific cases (using unpack *from Example 9.5.1):*

$$\begin{aligned} \{(2,13),(13,2)\} &\models (\mathsf{factor\text{-}alg}(26) \ggg \mathsf{unpack})(\square) \\ \{(2,6),(3,4),(4,3),(6,2),(12,1)\} &\models (\mathsf{factor\text{-}alg}(12) \ggg \mathsf{unpack})(\square) \end{aligned}$$

Instead of manually enumerating all possible outputs, we can make use of assert *to automatically check the result:*

```
check(n)  =  do (k, m) ← factor-alg(n)
                assert k ≥ 2
                assert k × m = n
```

$$\{\square\} \models (\mathsf{check}(26) \ggg \mathsf{unpack})(\square)$$

assert *either fails or returns* $\square$*, which is why we check for postcondition* $\{\square\}$*. To (theoretically) check all cases at once, we can write*

$$\{\square\} \models \left(\left(\begin{array}{ll} \textbf{do} & n \leftarrow \mathsf{pick_d}(\mathbb{Z}) \\ & (k,m) \leftarrow \mathsf{factor\text{-}alg}(n) \\ & \mathsf{assert}\ k \geq 2 \\ & \mathsf{assert}\ k \times m = n \end{array} \right) \ggg \mathsf{unpack} \right)(\square)$$

These are approximately the same steps taken by Featherweight VeriFast to verify a function:

- *Setting up a precondition (in our example, this consists of picking an arbitrary* n*).*
- *Executing the function under examination (calling* $\mathsf{factor\text{-}alg}(n)$*).*
- *Asserting the postcondition (the two assertions).*
- *Ensuring that no failure has occurred (checking that it satisfies postcondition* $\{\square\}$*).*

An alternative, more declarative way to factor an integer consists of magically picking the right results:

$$\begin{array}{rll} \mathsf{factor\text{-}dec}(n) = & \textbf{do} & k \leftarrow \mathsf{pick_d}(\mathbb{Z}) \\ & & m \leftarrow \mathsf{pick_d}(\mathbb{Z}) \\ & & \mathsf{assume}\ k \geq 2 \\ & & \mathsf{assume}\ k \times m = n \end{array}$$

The first two lines fork execution demonically, and the last two ensure that only those remain that have yielded correct results. In other words, all paths where $k < 2$ *or* $k \times m \neq n$ *have been "blocked" and do not influence further execution. Note how the behaviour of* factor-dec *is purely determined by* factor*'s contract.*

This approach of defining the same function in two different ways, once algorithmically, once declaratively, is key to modular verification: it allows each function to be verified in isolation, independently of the algorithmic implementation of other functions. We will elaborate on this in a later section.

Example 9.5.3. *Angelic choice allows us to elegantly represent failure and fork execution into multiple paths while only requiring one of them to succeed. This example shows how we can take further advantage of angelic choice to simplify the implementation of certain algorithms.*

During the formalisation of Featherweight VeriFast, we will need to find some element in a set satisfying a certain condition. If no such element can be found, failure must *ensue. If multiple such elements can be found, execution has to succeed for one of them. This clearly fits angelic choice.*

$$\begin{array}{rcl} \mathsf{find} & : & \mathcal{P}(A) \to (A \to \text{bool}) \to \mathit{Operator}_A^{\mathcal{S}\to\mathcal{S}} \\ \hline \mathsf{find}(A, P) & = & \textbf{do}\ a \leftarrow \mathsf{pick_a}(A) \\ & & \quad \mathsf{assert}\ P(a) \\ & & \quad \mathsf{return}\ a \end{array}$$

This should not be confused with the approach taken in Example 9.5.2.

- *Demonic choice and assumption express universal quantification:*

  $$\textbf{do}\ a \leftarrow \mathsf{pick_d}(A); \mathsf{assume}\ P(a); op(a)\ \textit{succeeds} \iff \forall\, a \in A.\ op(a)\ \textit{succeeds}$$

  If no such a exists, execution merely blocks, i.e. does not fail.

- *Angelic choice and assertion express existential quantification:*

  $$\textbf{do}\ a \leftarrow \mathsf{pick_a}(A); \mathsf{assert}\ P(a); op(a)\ \textit{succeeds} \iff \exists\, a \in A.\ op(a)\ \textit{succeeds}$$

  Execution fails if no such a exists.

Example 9.5.4. *Continuing from Example 9.3.4 (page 110), Fig. 9.3 shows how to formalise the semantics using the basic operators. We always operate on the same state, i.e. a store keeping track of the angelically chosen values:*

$$\mathcal{S} = \text{store} = \text{id} \to \mathbb{Z}$$

The operators store *and* set-store *get and set the store, respectively. Introducing them as separate operators (instead of just using* state *and* set-state *directly) improves abstraction: adding extra components to the state would not require changes to be made to the existing code.*

The lookup *operator looks up a variable's binding in the current store.* local *executes its second argument op in a local store s: after executing op, the store is restored to its old state. Lastly,* evaluate *defines how expressions must be evaluated.*

9.6 Result Algebra Models

This section discusses models of the result algebra, i.e. an interpretation for each operation that obeys the axioms as given in Def. 9.3.1 (page 110). We will mostly focus on their computability and their implementation in Coq.

$$\begin{array}{rcl}
\textsf{store} & : & \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}_{\text{store}} \\
\hline
\textsf{store} & = & \textbf{do}\ \ \textsf{state}
\end{array}$$

$$\begin{array}{rcl}
\textsf{set-store} & : & \text{store} \to \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}_{\text{unit}} \\
\hline
\textsf{set-store}(s) & = & \textbf{do}\ \ \textsf{set-state}(s)
\end{array}$$

$$\begin{array}{rcl}
\textsf{lookup} & : & \text{id} \to \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}_{\mathbb{Z}} \\
\hline
\textsf{lookup}(x) & = & \textbf{do}\ \ s \leftarrow \textsf{store} \\
 & & \phantom{\textbf{do}}\ \ \textsf{return}\ s(x)
\end{array}$$

$$\begin{array}{rcl}
\textsf{local} & : & \mathcal{S} \to \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}_{A} \to \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}_{A} \\
\hline
\textsf{local}(s, op) & = & \textbf{do}\ \ s' \leftarrow \textsf{store} \\
 & & \phantom{\textbf{do}}\ \ \textsf{set-store}(s) \\
 & & \phantom{\textbf{do}}\ \ r \leftarrow op \\
 & & \phantom{\textbf{do}}\ \ \textsf{set-store}(s') \\
 & & \phantom{\textbf{do}}\ \ \textsf{return}\ r
\end{array}$$

$$\begin{array}{rcl}
\textsf{evaluate} & : & \text{expression} \to \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}_{\mathbb{Z}} \\
\hline
\textsf{evaluate}(x) & = & \textbf{do}\ \ \textsf{lookup}(x) \\
\textsf{evaluate}(\textbf{pick}\ x \in X\ \textbf{in}\ e) & = & \textbf{do}\ \ n \leftarrow \textsf{pick}_{\text{a}}(X) \\
 & & \phantom{\textbf{do}}\ \ s \leftarrow \textsf{store} \\
 & & \phantom{\textbf{do}}\ \ \textsf{local}(s[x := n], \textsf{evaluate}(e)) \\
\textsf{evaluate}(\{A\}) & = & \textbf{do}\ \ \textsf{pick}_{\text{d}}(A) \\
\textsf{evaluate}(e_1 + e_2) & = & \textbf{do}\ \ v_1 \leftarrow \textsf{evaluate}(e_1) \\
 & & \phantom{\textbf{do}}\ \ v_2 \leftarrow \textsf{evaluate}(e_2) \\
 & & \phantom{\textbf{do}}\ \ \textsf{return}\ (v_1 + v_2) \\
\textsf{evaluate}(e_1 \times e_2) & = & \textbf{do}\ \ v_1 \leftarrow \textsf{evaluate}(e_1) \\
 & & \phantom{\textbf{do}}\ \ v_2 \leftarrow \textsf{evaluate}(e_2) \\
 & & \phantom{\textbf{do}}\ \ \textsf{return}\ (v_1 \times v_2)
\end{array}$$

Figure 9.3: Formalisation for Example 9.5.4

	Concrete	Semiconcrete	Symbolic
Infinite sets	✓	✓	
Computable			✓
Result Algebra	✓	✓	✓
Effective Result Algebra			✓

Figure 9.4: Computability Requirements

One main issue is the index set of the $\oplus$ and $\otimes$ operations. For the formalisation of Featherweight VeriFast, both finite and infinite indexing sets are needed. This causes some complications for the Coq implementation with regards to the type of the index set.

Using the Coq type Type gives total freedom: any well defined set is acceptable. But, while it does allow for computable index sets, it does not actually guarantee computability, which makes writing algorithms impossible in Coq[5].

Conversely, using the Coq type Set does promise computability, in which case infinite sets pose a problem. While coinductive types permit us to define computable infinite data structures, their use in algorithms is constrained so as to ensure termination.

Trying to find an ideal solution to these issues would result in a complex Coq implementation filled with obscure design choices which could only be motivated by elaborate and theoretical explanations. Instead, we have chosen for simplicity and split the Coq script in two parts.

The concrete and semiconcrete execution need not be computable and require infinite sets, while the symbolic execution should be computable and only operates on finite sets. This makes for a clean division: nowhere do we need computable infinite data structures. Thus, we can define two result algebras in Coq: one theoretical (noncomputable, infinite indexing sets) and one effective (computable, finite indexing sets). An overview is shown in Fig. 9.4.

Definitions and axioms for the noncomputable result algebra are shown in Listing 9.1 (page 111) and Listing 9.2 (page 112) and those for the effective result algebra[6] in Listing 9.3 and Listing 9.4. Multiple models have been written in Coq (see Fig. 9.5), two of which we discuss in detail and compare.

[5]Defining some extra axioms would make it possible, but could put consistency at risk.
[6]We refer the interested reader to Sect. C.6.2 for more clarification on the Coq code.

Description	Coq script	Effective
$\mathcal{S}^\star = \mathcal{P}(\mathcal{P}(\mathcal{S}))$	`SetOfSets.v`	
Weakest Preconditions	`WeakestPreconditions.v`	
Inductive Formulae	`InductiveFormulae.v`	
Inductive Formulae	`EInductiveFormulae.v`	✓

Figure 9.5: Coq Implementations of Result Algebra Models

```
Parameter RA : Set -> Set.

Variables
  (S : Set)
  (S_eqdec : forall s s' : S, { s = s' } + { s <> s' }).

Parameters (single     : S -> RA S)
           (models     : Ensemble S -> RA S -> Prop)
           (add mul    : list (RA S) -> RA S)
           (top bottom : RA S)
           (implies    : relation (RA S)
           (is_bottom  : RA S -> bool).
```

Listing 9.3: Effective Result Algebra Signature in Coq

9.6.1 Inductive Formulae

The inductive formulae model (Listing 9.5, found in the `InductiveFormulae.v` Coq script) is a rather straightforward model for the result algebra and makes the link with the boolean algebra more explicit: demonic choice corresponds to conjunction, angelic choice to disjunction. Top and bottom are modelled by true and false, respectively. The lack of a negation operator and the monotonicity axiom constitute the main difference between the result algebra and the boolean algebra. The result algebra, however, is a complete lattice: the order relation is $\Rrightarrow$ and the join and meet operations are $\oplus$ and $\otimes$, respectively. Readers unfamiliar with Coq can find more explanations in Sect. C.6.4.

```
Variable S : Set.

Axiom single_axiom :
  forall (s : S) (S : Ensemble S),
    models S (single s) <-> In s S.

Axiom top_axiom :
  equiv top (mul (empty_set (RA S))).

Axiom bottom_axiom :
  equiv bottom (add (empty_set (RA S))).

Axiom add_axiom :
  forall (Rs : list (RA S)) (S : Ensemble S),
    models S (add Rs) <-> exists R, set_In R Rs /\
                                    models S R.

Axiom mul_axiom :
  forall (Rs : list (RA S)) (S : Ensemble S),
    models S (mul Rs) <-> forall R, set_In R Rs ->
                                    models S R.

Axiom implies_axiom :
  forall (R R' : RA S),
    implies R R' <-> forall (S : Ensemble S),
                       models S R -> models S R'.

Axiom monotonic_models_axiom :
  forall (R : RA S) (S S' : Ensemble S),
    Included S' S -> models S' R -> models S R.
```

Listing 9.4: Effective Result Algebra Axioms in Coq

```
Inductive formula (S : Type) : Type :=
| f_and : forall I (R : I -> formula S), formula S
| f_or  : forall I (R : I -> formula S), formula S
| f_lit : S -> formula S.

Definition empty_R (S : Type) : False -> formula S.

Definition f_true  (S : Type) : formula S :=
  (@f_and S False (empty_R S)).

Definition f_false (S : Type) : formula S :=
  (@f_or S False (empty_R S)).

Definition RA (S : Type) : Type := formula S.

Variable S : Type.

Definition single (x : S) : RA S := f_lit x.

Fixpoint models (S : Ensemble S) (R : RA S) : Prop :=
  match R with
    | f_and J R' => forall j : J, models S (R' j)
    | f_or  J R' => exists j : J, models S (R' j)
    | f_lit s    => In s S
  end.

Definition add {I : Type} (R : I -> RA S) : RA S :=
  f_or R.

Definition mul {I : Type} (R : I -> RA S) : RA S :=
  f_and R.

Definition top    : RA S := f_true S.

Definition bottom : RA S := f_false S.

Definition implies (R R' : RA S) : Prop :=
  forall S, models S R -> models S R'.
```

Listing 9.5: Inductive Formulae Model in Coq

9.6.2 Weakest Preconditions

A particularly interesting model is the following[7]:

$$
\begin{aligned}
\mathcal{S}^\star &= \{\, R \mid R \in \mathcal{P}(\mathcal{P}(\mathcal{S})) \wedge \forall\, \Sigma, \Sigma'.\ \Sigma \in R \Rightarrow \Sigma \subseteq \Sigma' \Rightarrow \Sigma' \in R \,\} \\
[\![x]\!] &= \{\, S \in \mathcal{P}(\mathcal{S}) \mid x \in S \,\} \\
\bigoplus_{i \in I} R(i) &= \bigcup_{i \in I} R(i) \\
\bigotimes_{i \in I} R(i) &= \bigcap_{i \in I} R(i) \\
S \models R &= S \in R \\
\top &= \mathcal{P}(\mathcal{P}(\mathcal{S})) \\
\bot &= \emptyset
\end{aligned}
$$

In this section, we show how this model acts as a bridge between the result algebra and weakest preconditions (Sect. 5.4, page 49). We show how there is a correspondence between the manipulation of logical formulae representing weakest preconditions and the manipulation of results.

A result is an upward closed set of sets of states. Each of these sets $S \in R$ expresses a demonic choice over its elements $\bigotimes_{s \in S} [\![s]\!]$. The set of such sets corresponds to the angelic choice $\bigoplus_{S \in R} S$. Hence, the set of sets representation is actually a normalised form of the result.

Let us examine its relation to the weakest precondition verification conditions. As a reminder, the weakest precondition has type signature

$$\mathrm{wp} : \mathit{Command} \to \mathit{StatePredicate} \to \mathit{StatePredicate}$$

$\mathrm{wp}(c, Q)$ represents the precondition which has to be satisfied by an initial state so that execution will end up in a final state satisfying the postcondition Q. If we represent *StatePredicate*s by sets of states, the type signature becomes

$$\mathit{Command} \to \mathcal{P}(\mathcal{S}_{\text{post}}) \to \mathcal{P}(\mathcal{S}_{\text{pre}})$$

[7] We omitted the Coq script as its cryptic nature fails to provide any extra insights. We refer the interested reader to the `WeakestPreconditions.v` script file.

We have added the pre and post subscripts to make the distinction between preconditions and postconditions clearer. An equivalent form is

$$\mathit{Command} \to (\mathcal{S}_{\text{post}} \to \text{Prop}) \to \mathcal{S}_{\text{pre}} \to \text{Prop}$$

Flipping the second and third argument gives

$$\mathit{Command} \to \mathcal{S}_{\text{pre}} \to (\mathcal{S}_{\text{post}} \to \text{Prop}) \to \text{Prop}$$

which can be folded back into sets:

$$\mathit{Command} \to \mathcal{S}_{\text{pre}} \to \mathcal{P}(\mathcal{P}(\mathcal{S}_{\text{post}}))$$

In other words, given a command and an initial state, this reorganised wp function yields a set of sets of states, representing the resulting output states. This fits the mold of operators, i.e. the type can be written $\mathit{Command} \to \mathit{Operator}^{\mathcal{S}\to\mathcal{S}}$. It can act as an evaluation function exec, comparable to the one defined in Fig. 9.5.4. In short, this can be written

$$\text{wp}(c)^{\leftrightarrow} = \mathsf{exec}(c) \tag{9.2}$$

where $f^{\leftrightarrow} \equiv \lambda\ x\ y.\ f\ y\ x$ flips arguments.

As of yet, only the types have been shown to match. Let us fleshen out the link by considering operator binding, which corresponds to sequencing of commands.

$$\begin{array}{ccc}
\text{wp }(c_1; c_2)\ Q & = & \text{wp } c_1\ (\text{wp } c_2\ Q) \\
\shortparallel & & \shortparallel \\
(\mathsf{exec}\ (c_1; c_2))^{\leftrightarrow}\ Q & & (\mathsf{exec}\ c_1)^{\leftrightarrow}((\mathsf{exec}\ c_2)^{\leftrightarrow}\ Q) \\
\shortparallel & & \\
(\mathsf{exec}\ c_1 \ggg \mathsf{exec}\ c_2)^{\leftrightarrow}\ Q & &
\end{array}$$

The equality on the upper row originates from Def. 5.4.1 on page 49. The second row is obtained by using the equality from (9.2). Using transitivity and flipping arguments on both sides gives

$$\mathsf{exec}\ c_1 \ggg \mathsf{exec}\ c_2 = ((\mathsf{exec}\ c_1)^{\leftrightarrow} \circ (\mathsf{exec}\ c_2)^{\leftrightarrow})^{\leftrightarrow}$$

Intuitively, the flipping of arguments is similar to taking the inverse of the function (in this instance). The wp function works backwards, taking output states and returning input states, while exec operates in a forward direction, transforming input into output. Switching between these two modes of operation is achieved by flipping.

9.7 Operator Lemmas

The following sections will contain lemmas and theorems whose proofs we want to operate on the level of operators, i.e. if possible, we wish to abstract away underlying details. This section contains some lemmas which are essential for this.

First we divide operators into categories: one categorisation distinguishes operators based on which types of nondeterminism they exhibit, the other on how an operator interacts with state. Then we state some lemmas about how operators of certain categories interact with each other. An important result is the possibility of switching operators in bind operations if certain conditions are met.

Definition 9.7.1 (`RAOperatorTheorems.demonic_op`, `.demonic_op_K`). *An operator op is* demonic *if it always returns demonic results.*

$$\mathbf{demonic}(op) \equiv \forall\, \sigma.\ \mathbf{demonic}(op(\sigma)) \equiv \exists\, \mathcal{K}.\ op \overset{\cdot}{\Longleftrightarrow} \lambda\, \sigma.\ \bigotimes_{r \in \mathcal{K}(\sigma)} [\![r]\!]$$

The characteristic function *$\mathcal{K}(\sigma)$ returns the characteristic set of $op(\sigma)$.*

Definition 9.7.2 (`RAOperatorTheorems.angelic_op`, `.angelic_op_K`). *An operator is* angelic *if it always returns angelic results.*

$$\mathbf{angelic}(op) \equiv \forall\, \sigma.\ \mathbf{angelic}(op(\sigma)) \equiv \exists\, \mathcal{K}.\ op \overset{\cdot}{\Longleftrightarrow} \lambda\, \sigma.\ \bigoplus_{r \in \mathcal{K}(\sigma)} [\![r]\!]$$

The characteristic function *$\mathcal{K}(\sigma)$ returns the characteristic set of $op(\sigma)$.*

Definition 9.7.3 (`RAOperatorTheorems.deterministic_op`, `deterministic_op_K`). *An operator is* deterministic *if it always returns deterministic results.*

$$\mathbf{deterministic}(op) \equiv \forall\, \sigma.\ \mathbf{deterministic}(op(\sigma)) \equiv \exists\, \mathcal{K}.\ op \overset{\cdot}{\Longleftrightarrow} \lambda\, \sigma.\ [\![\mathcal{K}(\sigma)]\!]$$

The characteristic function *$\mathcal{K}(\sigma)$ returns the characteristic value of $op(\sigma)$.*

Definition 9.7.4 (`RAOperatorTheorems.pure_op`, `.pure_op_K`). *A* pure operator *op leaves the state untouched.*

$$\mathbf{pure}(op) \quad \equiv \quad \exists\, \mathcal{K}.\ op \overset{\cdot}{\Longleftrightarrow} \lambda\, \sigma.\ (\lambda\, r.\ (r, \sigma))^{\Uparrow}(\mathcal{K}(\sigma))$$

$\mathcal{K}$ is called the characteristic function *of the pure operator.*

Definition 9.7.5 (`RAOperatorTheorems.independent_op`, `.independent_op_K`). *A* state independent operator *op is a pure operator whose results do not depend on the state.*

$$\mathbf{independent}(op) \quad \equiv \quad \exists\, \mathcal{K}.\ op \overset{\cdot}{\Longleftrightarrow} \lambda\, \sigma.\ (\lambda\, r.\ (r, \sigma))^{\Uparrow}(\mathcal{K}(\sigma))$$

$\mathcal{K}$ is called the characteristic result *of the state independent operator.*

Lemma 9.7.1. *We can switch operators as follows:*

$$\left(\begin{array}{l}\mathbf{do}\ r_1 \leftarrow op_1\\ \quad\ r_2 \leftarrow op_2\\ \quad\ \mathsf{return}\ (r_1, r_2)\end{array}\right) \stackrel{\cdot}{\Longleftrightarrow} \left(\begin{array}{l}\mathbf{do}\ r_2 \leftarrow op_2\\ \quad\ r_1 \leftarrow op_1\\ \quad\ \mathsf{return}\ (r_1, r_2)\end{array}\right)$$

on the condition that one of the following combinations is true

- *op_1 is demonic and op_2 is state independent and demonic;*
- *op_1 is angelic and op_2 is state independent and angelic;*
- *op_1 is pure and angelic, op_2 is pure and angelic;*
- *op_1 is pure and demonic, op_2 is pure and demonic;*
- *op_1 is pure and demonic, op_2 is pure and demonic;*
- ...

Proof. We only prove the first case. The operators can be normalised as follows:

$$op_1 \stackrel{\cdot}{\Longleftrightarrow} \lambda\ \sigma.\ \bigotimes_{(r,\sigma')\in\mathcal{K}_1(\sigma)} [\![(r,\sigma')]\!] \tag{9.3}$$

$$op_2 \stackrel{\cdot}{\Longleftrightarrow} \lambda\ \sigma.\ \bigotimes_{r\in\mathcal{K}_2} [\![(r,\sigma)]\!] \tag{9.4}$$

Starting with the lhs

$$\begin{array}{l}\mathbf{do}\ r_1 \leftarrow op_1\\ \quad\ r_2 \leftarrow op_2\\ \quad\ \mathsf{return}\ (r_1, r_2)\end{array}$$

$$\stackrel{\cdot}{\Longleftrightarrow} op_1 \gg\!= (\lambda\ r_1.\ op_2 \gg\!= \lambda\ r_2.\ \mathsf{return}\ (r_1, r_2))$$

$$\stackrel{\cdot}{\Longleftrightarrow} op_1 \gg\!= (\lambda\ r_1.\ op_2 \gg\!= \lambda\ r_2.\ \lambda\ \sigma_2.\ [\![((r_1, r_2), \sigma_2)]\!])$$

Let $R(r_1)(r_2, \sigma) = [\![((r_1, r_2), \sigma)]\!]$

$$\stackrel{\cdot}{\Longleftrightarrow} op_1 \gg\!= (\lambda\ r_1.\ op_2 \ggg R(r_1))$$

$$\stackrel{\cdot}{\Longleftrightarrow} op_1 \gg\!= \left(\lambda\ r_1.\ R^{\Uparrow}(r_1) \circ op_2\right)$$

$$\stackrel{\cdot}{\Longleftrightarrow} op_1 \gg\!= \left(\lambda\ r_1.\ \lambda\ \sigma_1.\ R(r_1)^{\Uparrow}(op_2(\sigma_1))\right)$$

Rewriting op_2 using Equation (9.4),

$$\dot{\iff} op_1 \ggg\!= \left(\lambda\, r_1.\, \lambda\, \sigma_1.\, R(r_1)^{\Uparrow}\left(\bigotimes_{r_2 \in \mathcal{K}_2} [\![(r_2, \sigma_1)]\!]\right)\right)$$

$$\dot{\iff} op_1 \ggg\!= \left(\lambda\, r_1.\, \lambda\, \sigma_1.\, \bigotimes_{r_2 \in \mathcal{K}_2} R(r_1)(r_2, \sigma_1)\right)$$

$$\dot{\iff} op_1 \ggg \left(\lambda\, (r_1, \sigma_1).\, \bigotimes_{r_2 \in \mathcal{K}_2} [\![((r_1, r_2), \sigma_1)]\!]\right)$$

$$\dot{\iff} \left(\lambda\, (r_1, \sigma_1).\, \bigotimes_{r_2 \in \mathcal{K}_2} [\![((r_1, r_2), \sigma_1)]\!]\right)^{\Uparrow} \circ op_1$$

$$\dot{\iff} \lambda\, \sigma_0.\, \left(\lambda\, (r_1, \sigma_1).\, \bigotimes_{r_2 \in \mathcal{K}_2} [\![((r_1, r_2), \sigma_1)]\!]\right)^{\Uparrow} (op_1(\sigma_0))$$

Rewriting op_1 using Equation (9.4),

$$\dot{\iff} \lambda\, \sigma_0.\, \left(\lambda\, (r_1, \sigma_1).\, \bigotimes_{r_2 \in \mathcal{K}_2} [\![((r_1, r_2), \sigma_1)]\!]\right)^{\Uparrow} \left(\bigotimes_{(r_1, \sigma_1) \in \mathcal{K}_1(\sigma_0)} [\![(r_1, \sigma_1)]\!]\right)$$

$$\dot{\iff} \lambda\, \sigma_0.\, \bigotimes_{(r_1, \sigma_1) \in \mathcal{K}_1(\sigma_0)} \bigotimes_{r_2 \in \mathcal{K}_2} [\![((r_1, r_2), \sigma_1)]\!]$$

Now similarly for the rhs,

$$\begin{array}{ll} \textbf{do} & r_2 \leftarrow op_2 \\ & r_1 \leftarrow op_1 \\ & \textsf{return}\ (r_1, r_2) \end{array}$$

$$\dot{\iff} op_2 \ggg\!= \big(\lambda\, r_2.\, op_1 \ggg R'(r_2)\big)$$

where $R'(r_2)(r_1, \sigma) = [\![((r_1, r_2), \sigma_2)]\!]$

$$\dot{\iff} op_2 \ggg\!= (\lambda\, r_2.\, op_1 \ggg R'(r_2))$$

$$\dot{\iff} op_2 \ggg\!= \big(\lambda\, r_2.\, \lambda\, \sigma_1.\, R'(r_2)^{\Uparrow}(op_1(\sigma_1))\big)$$

Rewriting op_1 using Equation (9.3),

$$\dot{\iff} op_2 \ggg= \left(\lambda\ r_2.\ \lambda\ \sigma_1.\ R'(r_2)^{\Uparrow}\left(\bigotimes_{(r_1,\sigma_2)\in\mathcal{K}_1(\sigma_1)} [\![(r_1,\sigma_2)]\!]\right)\right)$$

$$\dot{\iff} op_2 \ggg \left(\lambda\ (r_2,\sigma_1).\ \left(\bigotimes_{(r_1,\sigma_2)\in\mathcal{K}_1(\sigma_1)} R'(r_2)(r_1,\sigma_2)\right)\right)$$

$$\dot{\iff} \lambda\ \sigma_0.\left(\lambda\ (r_2,\sigma_1).\ \left(\bigotimes_{(r_1,\sigma_2)\in\mathcal{K}_1(\sigma_1)} R'(r_2)(r_1,\sigma_2)\right)\right)^{\Uparrow}(op_2(\sigma_0))$$

Rewriting op_2 using Equation (9.4),

$$\dot{\iff} \lambda\ \sigma_0.\left(\lambda\ (r_2,\sigma_1).\ \left(\bigotimes_{(r_1,\sigma_2)\in\mathcal{K}_1(\sigma_1)} [\![((r_1,r_2),\sigma_2)]\!]\right)\right)^{\Uparrow}\left(\bigotimes_{r_2\in\mathcal{K}_2}[\![(r_2,\sigma_0)]\!]\right)$$

$$\dot{\iff} \lambda\ \sigma_0.\ \bigotimes_{r_2\in\mathcal{K}_2}\ \bigotimes_{(r_1,\sigma_2)\in\mathcal{K}_1(\sigma_0)} [\![((r_1,r_2),\sigma_2)]\!]$$

By Lemma 9.3.20

$$\dot{\iff} \lambda\ \sigma_0.\ \bigotimes_{(r_1,\sigma_2)\in\mathcal{K}_1(\sigma_0)}\ \bigotimes_{r_2\in\mathcal{K}_2} [\![((r_1,r_2),\sigma_2)]\!]$$

□

Lemma 9.7.2 (`RAOperatorTheorems.bind_demonic`, `.bind_angelic`, `.bind_deterministic`, `.bind_pure`, `.bind_independent`)**.** *Binding preserves an operator's property of being demonic, angelic and deterministic. Using the following shorthand notation*

$$P_\forall(f) = \forall\ x.\ P(f\ x)$$

$$\begin{array}{rcrcr}
\textbf{demonic}(op) & \Rightarrow & \textbf{demonic}_\forall(f) & \Rightarrow & \textbf{demonic}(op \ggg= f) \\
\textbf{angelic}(op) & \Rightarrow & \textbf{angelic}_\forall(f) & \Rightarrow & \textbf{angelic}(op \ggg= f) \\
\textbf{deterministic}(op) & \Rightarrow & \textbf{deterministic}_\forall(f) & \Rightarrow & \textbf{deterministic}(op \ggg= f) \\
\textbf{pure}(op) & \Rightarrow & \textbf{pure}_\forall(f) & \Rightarrow & \textbf{pure}(op \ggg= f) \\
\textbf{independent}(op) & \Rightarrow & \textbf{independent}_\forall(f) & \Rightarrow & \textbf{independent}(op \ggg= f)
\end{array}$$

where

$$\begin{array}{rcl}
op & : & Operator_A^{\mathcal{S}_1\to\mathcal{S}_2} \\
f & : & A \to Operator_B^{\mathcal{S}_2\to\mathcal{S}_3}
\end{array}$$

9.8 Concrete Execution

As explained in Sect. 9.2, we will define three semantics for the Small Imperative Language defined in Sect. 9.1. This section fully formalises the first, known as the concrete execution.

9.8.1 Formalisation

We start with posing ourselves the question: Given a SIL program, how do we ascertain that no failures occur during execution? The most straightforward approach would be to simply execute the program until it finishes. We remind the reader of some key properties of the SIL language:

- All heap accesses must be checked: invalid accesses must lead to failure.
- No guarantees are made regarding the location nor contents of newly allocated memory blocks. Verification should take into account all possible locations and contents.

Modelling these will pose no problem as we have prepared a semantic framework specially for this purpose: the concrete semantics will take the form of an operator, as described in Sect. 9.4.

Operators require us to define a state. In the case of the concrete execution, we need it to comprise two components:

- A store which keeps track of variable bindings. The only allowed values are natural numbers. Thus, a store is modelled as a (total) function from variables to $\mathbb{N}$.
- A heap, represented as a multiset of heap chunks, similar to those that have already been discussed in Sect. 8.2.1. The concrete execution uses two kinds of heap chunks:
  - $p \mapsto v$ expresses that the memory cell at address p contains value v. Reading and writing to this memory location is only allowed if the corresponding heap chunk is present on the heap.
  - $\mathsf{mb}(p, n)$ serves to keep track of memory cells that constitute a memory block. For example, if a $\mathsf{mb}(10, 2)$ chunk is present on the heap, so will $10 \mapsto v_1$ and $11 \mapsto v_2$, for some $v_1, v_2 \in \mathbb{N}$. The command **free**(10) will rely on the $\mathsf{mb}(10, 2)$ chunk to determine how many $\mapsto$ chunks it needs to remove from the heap.

We call this a *concrete state*.

Definition 9.8.1 (`ConcreteExecution.CStore.t`). *A* concrete store *s maps variable identifiers to values.*

$$CStore \equiv Id \rightarrow \mathbb{N}$$

Definition 9.8.2 (`ConcreteExecution.zero_store`). *The* zero concrete store *maps every variable to* 0*.*

$$s_0 \equiv \lambda\, x.\, 0$$

Definition 9.8.3 (`ConcreteExecution.Predicate.t`). *A* concrete predicate *p is either the points-to predicate* $\mapsto$ *or the malloc block predicate* mb*.*

$$CPredicate \equiv \{\mapsto, \mathsf{mb}\}$$

Definition 9.8.4 (`ConcreteExecution.CChunk.t`). *A* concrete chunk α *is the application of a concrete predicate to two arguments belonging to* $\mathbb{N}$*.*

$$CChunk \equiv \{\, p(x,y) \mid p \in CPredicate,\ x,y \in \mathbb{N}\,\}$$

Definition 9.8.5 (`ConcreteExecution.CHeap.t`). *A* concrete heap *h is a multiset of concrete chunks, i.e. total functions from concrete chunks to naturals.*

$$CHeap \equiv CChunk \rightarrow \mathbb{N}$$

The reader might wonder why the heap is defined as a multiset instead of a normal set. It is certainly so that in the case of the concrete execution, having the same chunk appear more than once on the heap makes no sense. However, the other executions will perform heap abstraction (introduced in Sect. 9.9) which will make it necessary to be able to have the heap contain the same chunk multiple times. For reasons of consistency, we have chosen to make the concrete heap a multiset also.

Definition 9.8.6 (`ConcreteExecution.concrete_state`). *A* concrete execution state σ *consists of a concrete store and a concrete heap.*

$$CState \equiv CStore \times CHeap$$

We introduce the following notation for a state consisting of a store s and a heap h: $\langle s,h\rangle_{\mathrm{c}}$*.*

SIL supports expressions and boolean expressions. When needed, they are evaluated in the current store. For completeness, we formalise how to perform this evaluation.

Definition 9.8.7 (`Expression.evaluate`). *Evaluating an expression e in a concrete store s is defined as*

$$\begin{array}{rcl} \text{eval-expr} & : & \mathit{CStore} \to \mathit{Expression} \to \mathbb{N} \\ \hline \text{eval-expr}(s, n) & = & n \\ \text{eval-expr}(s, x) & = & s(x) \\ \text{eval-expr}(s, e +_{\mathrm{e}} e') & = & \text{eval-expr}(s, e) + \text{eval-expr}(s, e') \\ \text{eval-expr}(s, e -_{\mathrm{e}} e') & = & \text{eval-expr}(s, e) - \text{eval-expr}(s, e') \\ \text{eval-expr}(s, e \times_{\mathrm{e}} e') & = & \text{eval-expr}(s, e) \times \text{eval-expr}(s, e') \end{array}$$

Definition 9.8.8 (`BooleanExpression.evaluate`). *Evaluating a boolean expression e in a concrete store s is defined as*

$$\begin{array}{rcl} \text{eval-bexpr} & : & \mathit{CStore} \to \mathit{BExpression} \to \mathit{Bool} \\ \hline \text{eval-bexpr}(s, \mathbf{true}_{\mathrm{b}}) & = & \text{true} \\ \text{eval-bexpr}(s, e =_{\mathrm{b}} e') & = & \text{eval-expr}(s, e) = \text{eval-expr}(s, e') \\ \text{eval-bexpr}(s, e <_{\mathrm{b}} e') & = & \text{eval-expr}(s, e) < \text{eval-expr}(s, e') \\ \text{eval-bexpr}(s, e \leq_{\mathrm{b}} e') & = & \text{eval-expr}(s, e) \leq \text{eval-expr}(s, e') \\ \text{eval-bexpr}(s, b \wedge_{\mathrm{b}} b') & = & \text{eval-expr}(s, b) \wedge \text{eval-expr}(s, b') \\ \text{eval-bexpr}(s, \neg_{\mathrm{b}}\, b) & = & \neg\, \text{eval-bexpr}(s, b) \end{array}$$

We now proceed with the definition of a number of auxiliary operators. Since all operate on the same state type *CStore* and do not modify it, we define a shorthand notation so as to make the type signatures less cluttered.

Definition 9.8.9 (`ConcreteExecution.c_operator`). *A concrete operator is an operator operating on concrete states.*

$$\mathit{COperator}_A \equiv \mathit{Operator}_A^{\mathit{CState} \to \mathit{CState}}$$

The operators current-store and set-current-store serve to get and set the store in the current state. Similarly, current-heap and set-current-heap are provided as low level operators to interact with the heap.

Definition 9.8.10 (`current store`, `set_current store`). *The operator* current-store *fetches the store from the current program state.*

$$\begin{array}{rcl} \text{current-store} & : & \mathit{COperator}_{\mathit{CStore}} \\ \hline \text{current-store} & = & \begin{array}{l} \mathbf{do}\ \langle s, h \rangle_{\mathrm{c}} \leftarrow \text{current-state} \\ \quad\ \ \text{return } s \end{array} \end{array}$$

The operator set-current-store(s') *sets* s' *as the new store.*

$$\begin{array}{rcl} \text{set-current-store} & : & \mathit{CStore} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{set-current-store}(s') & = & \begin{array}{l} \mathbf{do}\ \langle s, h \rangle_{\mathrm{c}} \leftarrow \text{current-state} \\ \quad\ \ \text{set-current-state}(\langle s', h \rangle_{\mathrm{c}}) \end{array} \end{array}$$

Definition 9.8.11 (`current heap`, `set_current heap`). current-heap *fetches the heap from the current program state.*

$$\begin{array}{rcl} \text{current-heap} & : & \mathit{COperator}_{\mathit{CHeap}} \\ \hline \text{current-heap} & = & \textbf{do}\ \langle s, h\rangle_{\text{c}} \leftarrow \text{current-state} \\ & & \quad\ \text{return } h \end{array}$$

The operator set-current-heap(h') *sets* h' *as the new heap.*

$$\begin{array}{rcl} \text{set-current-heap} & : & \mathit{CHeap} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{set-current-heap}(h') & = & \textbf{do}\ \langle s, h\rangle_{\text{c}} \leftarrow \text{current-state} \\ & & \quad\ \text{set-current-state}(\langle s, h'\rangle_{\text{c}}) \end{array}$$

We build higher level operators on top of current-store and set-current-store: read-store and update-store look up and modify a variable binding in the store, respectively. with-store(s, op) executes op locally in the given store s: it is similar to creating a new stack frame in C.

Definition 9.8.12 (`ConcreteExecution.read_store`). *The operator* read-store(x) *looks up the value to which x is bound in the store.*

$$\begin{array}{rcl} \text{read-store} & : & \mathit{Var} \to \mathit{COperator}_{\mathbb{N}} \\ \hline \text{read-store}(x) & = & \textbf{do}\ s \leftarrow \text{current-store} \\ & & \quad\ \text{return } s(x) \end{array}$$

Definition 9.8.13 (`ConcreteExecution.update_store`). update-store(x, v) *updates the store so that x is mapped to v, and all other variables' mappings remain unchanged. This operator never fails.*

$$\begin{array}{rcl} \text{update-store} & : & \mathit{Var} \to \mathbb{N} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{update-store}(x, v) & = & \textbf{do}\ s \leftarrow \text{current-store} \\ & & \quad\ \text{set-current-store}(s[x := v]) \end{array}$$

Example 9.8.1. *Intuitively,* read-store(x) *executed in the state* $\sigma = \langle s_0[\mathrm{x} := 4], \emptyset\rangle_{\text{c}}$ *leaves the state unchanged and yields* 4 *as return value. More formally, the actual result is*

$$\text{read-store}(\mathrm{x})(\sigma) \Longleftrightarrow [\![(4, \sigma)]\!]$$

where 4 *is the return value and* σ *the output state. Thus, we also have*

$$\{(4, \sigma)\} \models \text{read-store}(\mathrm{x})(\sigma)$$

Writing the store gives:

$$\text{update-store}(\mathrm{x}, 5)(\sigma) \Longleftrightarrow [\![(\square, \langle s_0[\mathrm{x} := 5], \emptyset\rangle_{\text{sc}})]\!]$$

Definition 9.8.14 (`ConcreteExecution.with_store`). *The operator* with-store(s, op) *saves the current store, sets s as the new current store, applies op, and restores the original store.*

$$\begin{array}{rcl} \text{with-store} & : & \mathit{CStore} \to \mathit{COperator}_A \to \mathit{COperator}_A \\ \hline \text{with-store}(s, op) & = & \textbf{do}\ s' \leftarrow \text{current-store} \\ & & \quad \text{set-current-store}(s) \\ & & \quad r \leftarrow op \\ & & \quad \text{set-current-store}(s') \\ & & \quad \text{return}\ r \end{array}$$

The evaluate operator acts a wrapper around the function eval-expr.

Definition 9.8.15 (`ConcreteExecution.evaluate`). evaluate(e) *evaluates the expression e in the current store.*

$$\begin{array}{rcl} \text{evaluate} & : & \mathit{Expression} \to \mathit{COperator}_{\mathbb{N}} \\ \hline \text{evaluate}(e) & = & \textbf{do}\ s \leftarrow \text{current-store} \\ & & \quad \text{return eval-expr}(s, e) \end{array}$$

The assume-bexpr and assert-bexpr evaluate a boolean expression in the current store. If the boolean expression does not evaluate to true, execution blocks or fails, respectively.

Definition 9.8.16 (`ConcreteExecution.assume_bexpr`). *The* assume-bexpr *operator prunes the program states which do not satisfy a certain condition.*

$$\begin{array}{rcl} \text{assume-bexpr} & : & \mathit{BExpression} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{assume-bexpr}(b) & = & \textbf{do}\ s \leftarrow \text{current-store} \\ & & \quad \text{assume}(\text{eval-bexpr}(s, b) = \text{true}) \end{array}$$

Definition 9.8.17 (`ConcreteExecution.assert_bexpr`). *The* assert-bexpr *operator demands that a certain condition be true. If this is not the case, failure ensues.*

$$\begin{array}{rcl} \text{assert-bexpr} & : & \mathit{BExpression} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{assert-bexpr}(b) & = & \textbf{do}\ s \leftarrow \text{current-store} \\ & & \quad \text{assert}(\text{eval-bexpr}(s, b) = \text{true}) \end{array}$$

After having defined store related operators, we now turn our attention to the heap.

Definition 9.8.18 (`ConcreteExecution.produce_chunk`). produce-chunk(α) *adds a single chunk α to the heap. This operator never fails.*

$$\begin{array}{rcl} \text{produce-chunk} & : & \mathit{CChunk} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{produce-chunk}(\alpha) & = & \textbf{do}\ h \leftarrow \text{current-heap} \\ & & \quad \text{set-current-heap}(\ h \uplus \{\alpha\}\ ) \end{array}$$

Definition 9.8.19 (`ConcreteExecution.consume_chunk`). consume-chunk(α) *removes a single chunk α from the heap. If the heap does not contain the given chunk, failure ensues. If the heap contains the given chunk multiple times, one instance is removed.*

$$\begin{array}{rcl}
\text{consume-chunk} & : & \mathit{CChunk} \to \mathit{COperator}_{\text{unit}} \\ \hline
\text{consume-chunk}(\alpha) & = & \textbf{do}\ h \leftarrow \text{current-heap} \\
 & & \qquad \text{assert}\ \alpha \in h \\
 & & \qquad \text{set-current-heap}(h - \{\alpha\})
\end{array}$$

Definition 9.8.20 (`ConcreteExecution.alloc_set`, `.allocate`). allocate(n) *allocates a memory block of size n on the heap.*

$$\begin{array}{rcl}
\text{alloc-set} & : & \mathit{COperator}_{\mathcal{P}(\mathbb{N})} \\ \hline
\text{alloc-set} & = & \textbf{do}\ h \leftarrow \text{current-heap} \\
 & & \qquad \text{return}\ \{\ \ell \mid \ell \mapsto v \in h\ \}
\end{array}$$

$$\begin{array}{rcl}
\text{allocate-at} & : & \mathbb{N} \to \mathbb{N}^{[*]} \to \mathit{COperator}_{\text{unit}} \\ \hline
\text{allocate-at}\ \ell\ [\,] & = & \textbf{do}\ \text{nop} \\
\text{allocate-at}\ \ell\ (v :: vs) & = & \textbf{do}\ A \leftarrow \text{alloc-set} \\
 & & \qquad \text{assume}((\ell + \|vs\|) \notin A) \\
 & & \qquad \text{produce-chunk}((\ell + \|vs\|) \mapsto v) \\
 & & \qquad \text{allocate-at}\ \ell\ vs
\end{array}$$

$$\begin{array}{rcl}
\text{allocate} & : & \mathbb{N} \to \mathit{COperator}_{\mathbb{N}} \\ \hline
\text{allocate}(n) & = & \textbf{do}\ (\ell :: vs) \leftarrow \text{pick}_{\text{d}}(\mathbb{N}^{[n+1]}) \\
 & & \qquad \text{allocate-at}\ \ell\ vs \\
 & & \qquad \text{return}\ \ell
\end{array}$$

where $A^{[*]}$ denotes the set of lists of arbitrary length with elements of type A, $A^{[n]}$ denotes the set of lists of length n with elements of type A and $\|xs\|$ denotes the length of the list xs.

Memory allocation is the sole source of nondeterminism in the concrete execution. Allocating a memory block of size n consists of the following steps:

1. We start with picking[8] a random memory location ℓ and n random values vs. Whether the memory block can fit at location ℓ will be determined later.

2. allocate-at(ℓ, vs) produces $\|vs\|$ consecutive $\mapsto$ chunks, starting at ℓ. The operator assume($(\ell + \|vs\|) \notin A$) makes sure the location is not already

[8]We pregenerate all random values in one step to simplify proofs later.

in use. If it is, execution blocks: we are not interested in simulating overlapping memory blocks. Each memory cell is assigned one of the random values in the list *vs*.

3. The memory location ℓ is returned.

It is sometimes necessary to find a chunk on the heap whose arguments are not all known. For example, reading from memory location ℓ requires a chunk $\ell \mapsto v$ to be present on the heap, where v is unknown. Finding this chunk must be done angelically: if no matching chunks are found, failure must ensue. Multiple matches are not possible in the concrete execution, as memory is always allocated on unused parts of the heap.

Definition 9.8.21 (`ConcreteExecution.find_chunk`)**.** find-chunk(p, x) *returns a chunk with the specified predicate p and first argument xs.*

$$\begin{array}{rcl}
\textsf{find-chunk} & : & \mathit{CPredicate} \to \mathbb{N} \to \mathit{COperator}_{\mathit{CChunk}} \\
\hline
\textsf{find-chunk}(p, x) & = & \textbf{do}\ \ h \leftarrow \textsf{current-heap} \\
& & \quad\ \ p'(x', y') \leftarrow \textsf{pick}_{\textsf{a}}(h) \\
& & \quad\ \ \textsf{assert}(p = p') \\
& & \quad\ \ \textsf{assert}(x = x') \\
& & \quad\ \ \textsf{return}\ p(x, y')
\end{array}$$

The following operator is allocate-at's opposite: consume-cell(ℓ, n) removes n consecutive $\mapsto$ chunks from the heap, starting at ℓ.

Definition 9.8.22 (`ConcreteExecution.consume_chunk`)**.** consume-cells(ℓ, n) *removes the following n chunks from the heap: $\ell + i \mapsto v$ with $i = 0, \ldots, n-1$.*

$$\begin{array}{rcl}
\textsf{consume-cells} & : & \mathbb{N} \to \mathbb{N} \to \mathit{COperator}_{\mathrm{unit}} \\
\hline
\textsf{consume-cells}(\ell, 0) & = & \textbf{do}\ \ \textsf{nop} \\
\textsf{consume-cells}(\ell, n+1) & = & \textbf{do}\ \ \alpha \leftarrow \textsf{find-chunk}(\mapsto, \ell + n) \\
& & \quad\ \ \textsf{consume-chunk}(\alpha) \\
& & \quad\ \ \textsf{consume-cells}(\ell, n)
\end{array}$$

Definition 9.8.23 (`ConcreteExecution.read_cell`)**.** *The operator* read-cell(ℓ) *looks for a $\ell \mapsto v$ chunk on the heap and extracts the value v. Failures ensues if no such chunk can be found.*

$$\begin{array}{rcl}
\textsf{read-cell} & : & \mathbb{N} \to \mathit{COperator}_{\mathbb{N}} \\
\hline
\textsf{read-cell}(\ell) & = & \textbf{do}\ \ (\ell \mapsto v) \leftarrow \textsf{find-chunk}(\mapsto, \ell) \\
& & \quad\ \ \textsf{return}\ v
\end{array}$$

Definition 9.8.24 (ConcreteExecution.write_cell). *The operator* write-cell(ℓ, v) *replaces a* $\ell \mapsto v'$ *chunk by* $\ell \mapsto v$. *If no such chunk is present on the heap, failure ensues.*

$$\begin{array}{rcl} \text{write-cell} & : & \mathbb{N} \to \mathbb{N} \to \mathit{COperator}_{\text{unit}} \\ \hline \text{write-cell}(\ell, v) & = & \textbf{do}\ \alpha \leftarrow \text{find-chunk}(\mapsto, \ell) \\ & & \quad\ \ \text{consume-chunk}(\alpha) \\ & & \quad\ \ \text{produce-chunk}(\ell \mapsto v) \end{array}$$

Definition 9.8.25 (ConcreteExecution.block_size). *The operator* block-size(ℓ) *looks for a* mb(ℓ, n) *chunk on the heap and extracts its size n. This operator leads to failure if no such chunk is present.*

$$\begin{array}{rcl} \text{block-size} & : & \mathbb{N} \to \mathit{COperator}_{\mathbb{N}} \\ \hline \text{block-size}(\ell) & = & \textbf{do}\ \text{mb}(\ell, n) \leftarrow \text{find-chunk}(\text{mb}, \ell) \\ & & \quad\ \ \text{return}\ n \end{array}$$

Definition 9.8.26 (ConcreteExecution.concrete_execution_n, .concrete_execution). *The operator* c-execute(c) *denotes the concrete execution of the given command c. We define it by induction on the execution depth. Specifically, we first define the step-indexed operator* c-execute$_n(c)$ *(see Figure 9.6), which denotes the execution of command c up to depth n.* c-execute$_n(c)$ *contains only the execution paths whose depth is at most n. We can then define* c-execute(c)*:*

$$\text{c-execute}(c) = \bigotimes_{n \in \mathbb{N}} \text{c-execute}_n(c)$$

We sometimes use the abbreviation c-exec *instead of* c-execute.

Definition 9.8.27 (ConcreteExecution.valid_program). *A program P is* valid *if, for any input argument and empty heap, execution does not end in failure:*

$$\begin{array}{rcl} \text{verify} & = & \textbf{do}\ v \leftarrow \text{pick}_{\text{d}}(\mathbb{N}) \\ & & \quad\ \ \text{update-store}(x, v) \\ & & \quad\ \ \text{c-execute}(c) \end{array}$$

$$\mathit{valid\text{-}program}_{\text{c}}(P) \quad \equiv \quad \text{verify}(\langle s_0, \emptyset \rangle_{\text{c}}) \not\Longleftrightarrow \bot$$

where c is the body of the main routine and x its parameter.

9.8.2 Shortcomings

While the concrete execution is conceptually simple, it is not computable. Let us consider execution trees of a program. Giving an exact description of how to

$$
\begin{array}{rl}
\text{c-exec}_0(c) = & \textbf{do}\ \text{stop} \\[1em]
\text{c-exec}_{n+1}(x := e) = & \textbf{do}\ v \leftarrow \text{evaluate}(e) \\
 & \phantom{\textbf{do}}\ \text{update-store}(x, v) \\[1em]
\text{c-exec}_{n+1}(x := [e]) = & \textbf{do}\ \ell \leftarrow \text{evaluate}(e) \\
 & \phantom{\textbf{do}}\ v \leftarrow \text{read-cell}(\ell) \\
 & \phantom{\textbf{do}}\ \text{update-store}(x, v) \\[1em]
\text{c-exec}_{n+1}([e] := e') = & \textbf{do}\ \ell \leftarrow \text{evaluate}(e) \\
 & \phantom{\textbf{do}}\ v \leftarrow \text{evaluate}(e') \\
 & \phantom{\textbf{do}}\ \text{write-cell}(\ell, v) \\[1em]
\text{c-exec}_{n+1}(c; c') = & \textbf{do}\ \text{c-exec}_n(c) \\
 & \phantom{\textbf{do}}\ \text{c-exec}_n(c') \\[1em]
\text{c-exec}_{n+1}(\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c') = & \textbf{do}\ op \leftarrow \text{pick}_{\text{d}}\left(\left\{\begin{array}{l} \text{A}(b) \gg \text{c-exec}_n(c) \\ \text{A}(\neg_{\text{b}} b) \gg \text{c-exec}_n(c') \end{array}\right\}\right) \\
 & \phantom{\textbf{do}}\ op \\[1em]
\text{c-exec}_{n+1}(r(e)) = & \textbf{do}\ v \leftarrow \text{evaluate}(x) \\
 & \phantom{\textbf{do}}\ \text{with-store}(s_0[x := v], \text{c-exec}_n(c)) \\
 & \text{where } \textbf{routine}\ r(x) = c \\[1em]
\text{c-exec}_{n+1}(x := \textbf{malloc}(n)) = & \textbf{do}\ \ell \leftarrow \text{allocate}(n) \\
 & \phantom{\textbf{do}}\ \text{produce-chunk}(\text{mb}(\ell, n)) \\
 & \phantom{\textbf{do}}\ \text{update-store}(x, \ell) \\[1em]
\text{c-exec}_{n+1}(\textbf{free}(e)) = & \textbf{do}\ \ell \leftarrow \text{evaluate}(e) \\
 & \phantom{\textbf{do}}\ n \leftarrow \text{block-size}(\ell) \\
 & \phantom{\textbf{do}}\ \text{consume-chunk}(\text{mb}(\ell, n)) \\
 & \phantom{\textbf{do}}\ \text{consume-cells}(\ell, n) \\[1em]
\text{c-exec}_{n+1}(\textbf{skip}) = & \textbf{do}\ \text{return}\ \square
\end{array}
$$

$$\text{A} = \text{assume-bexpr}$$

Figure 9.6: Concrete Execution

build an execution tree is difficult due to the different kinds of nondeterminism. Fortunately, for our purposes, only a sketch is necessary. Let us assume that the execution tree is as deep as there are computational steps and that a node has as many children as the number of subpaths execution forks into at that points forks. For example, $\mathsf{pick}_{\mathsf{d}}(\{1, 2, 3\})$ leads a node to have three children.

Verification as we described it then consists of building the entire tree and ensuring that certain conditions are met. For this to be computable, the tree should be of finite size. In the case of concrete execution, this is not so:

- Allocation forks into an infinite number of paths, due to the location and contents of each memory cell to be picked nondeterministically from $\mathbb{N}$. Thus, the tree's branching factor is infinite.
- The execution tree could have infinite depth in case execution never ends, e.g. due to endless recursion.

Thus, the tree is both infinitely wide and deep. In the next section, we define the semiconcrete execution, which eliminates the infinite depth. In a second step, the symbolic execution takes care of the infinite breadth.

9.9 Semiconcrete Execution

This section discusses the semiconcrete execution, which partly solves the shortcomings of the concrete execution, as discussed in Sect. 9.8.2. A full formalisation is presented in Sect. 9.9.1, after which we relate it to the concrete execution in Sect. 9.9.2.

9.9.1 Formalisation

To solve the issue of the infinitely deep execution trees, the semiconcrete execution modularises verification. When a routine invocation is encountered, the concrete execution (Fig. 9.6) proceeds to execute the routine's body. The semiconcrete execution takes a different approach: it will directly transform the program state based on the routine's contract. Example 9.5.3 gave a preview of this: it explained how instead of actually computing a result (such as the concrete execution does), one can also nondeterministically pick values that obey the contract.

For this approach to work, routines must be equipped with contracts. SIL needs to be extended in a number of ways, giving rise to SIL^{++}. To express contracts, we introduce *assertions*.

Definition 9.9.1 (`Assertion.t`). *An* assertion *a is defined as*

$$\begin{array}{lcl} \mathit{Assertion} & ::= & \mathit{BExpr} \\ & | & \mathit{Predicate}(\mathit{Expr}, ?\mathit{Id}) \\ & | & \mathit{Assertion} \star \mathit{Assertion} \\ & | & \textbf{if } \mathit{BExpr} \textbf{ then } \mathit{Assertion} \textbf{ else } \mathit{Assertion} \end{array}$$

For simplicity, the language contains only predicate assertions whose first argument is an expression and whose second argument is a variable pattern*, which binds the value of the corresponding argument in the matched chunk to the specified variable x.*

As an exception to the notation for assertions, we will use the special syntax $e \mapsto ?x$ to denote cells; the reader can imagine it stands for $\mapsto (e, ?x)$. An example of assertions is given shortly, in Example 9.9.1.

Definition 9.9.2 (`SILPP.routine_definition`). Routine specifications *consist of a precondition and a postcondition, both of which are assertions. We introduce the following new syntax:*

$$\textbf{routine } r(x) \textbf{ requires } \mathit{Assertion} \textbf{ ensures } \mathit{Assertion} = \mathit{Command}$$

Example 9.9.1. *The following C code*

```
void isqrt(int *p) { ... }
```

can be implemented in SIL^{++} as follows

$$\begin{array}{l} \textbf{routine } \mathrm{isqrt}(p) \\ \quad \textbf{requires } p \mapsto ?v \;\star\; 0 \leq v \\ \quad \textbf{ensures } p \mapsto ?w \;\star\; w \times w \leq v \;\star\; v \leq (w+1) \times (w+1) \\ = \ldots \end{array}$$

Note how we use variable patterns (?v and ?w) to capture the contents of cells. A variable's scope extends until the end of the contract. Variables bound in the precondition can thus be used in the postcondition so as to relate the results to the inputs.

Verification using the concrete execution entails the building of one execution tree and just making sure that it only fails where allowed (some failures are allowed due to angelic choice). Using the semiconcrete execution consists of building many such trees, one for each routine. Each tree is then checked separately in turn, not only for failures but also to make sure the contract is upheld. Modularity is reflected by the fact that changing the body of one routine does not affect the verification result of other routines, as these only depend on the routine's contract.

Technically, this is all that is necessary to eliminate the possibility of endless recursion and thus reduce the depth of execution trees to finite size. However, the expressiveness of contracts is severely limited: we can only express the contracts of trivial routines. This issue has already been discussed in Sect. 8.2.3: currently, we cannot write down contracts for routines that work on data structures of arbitrary size. One solution is to introduce heap abstraction in the form of user defined predicates.

Definition 9.9.3 (`SILPP.predicate_definition`). *A* predicate definition *associates a predicate name p with a parameter list (fixed to a length of two, for simplicity), and an assertion as body:*

$$\mathit{PredicateDefinition} \; ::= \; \textbf{predicate}\; \mathit{PredicateName}(\mathit{Id}, \mathit{Id}) = \mathit{Assertion}$$

where $\mathit{PredicateName} = \{\mapsto, \mathsf{mb}\} \cup \mathit{UserDefined}$*.*

Definition 9.9.4 (`SILPP.program`, `.wellformed_program`). *A SIL*$^{++}$ program *is a set of routines, one of which is the main routine and is named* main*, and a set of predicate definitions. We require that every routine call refers to a routine defined in the program. We also require that every predicate mentioned in the code has a definition.*

Example 9.9.2. *A list predicate can be defined in SIL*$^{++}$ *as follows:*

$$\begin{array}{l}
\textbf{predicate}\ \mathrm{List}(p, n) = \\
\quad \textbf{if}\ p =_{\mathrm{b}} 0 \\
\quad\ \textbf{then}\ n =_{\mathrm{b}} 0 \\
\quad\ \textbf{else}\ p \mapsto ?v \,\star\, (p +_{\mathrm{e}} 1) \mapsto ?q \,\star\, \mathsf{mb}(p, 2) \,\star\, \mathrm{List}(q, ?m) \,\star\, n =_{\mathrm{b}} m +_{\mathrm{e}} 1
\end{array}$$

Like VeriFast, Featherweight VeriFast requires explicit folding and unfolding of custom chunks (i.e. chunks using user defined predicates). We add two new commands for this purpose.

Definition 9.9.5 (`SILPP.command`). *SIL*$^{++}$ *offers two new commands which allow us to open (or unfold) and close (or fold) custom chunks:*

$$c ::= \ldots \mid \textbf{open}\ \mathit{PredicateName}(\mathit{Expr}, \_) \mid \textbf{close}\ \mathit{PredicateName}(\mathit{Expr}, \mathit{Expr})$$

Note that in the case of the **open** command, the second argument of the heap chunk is never given. The semiconcrete execution will look for a chunk on the heap with matching predicate and first argument. If multiple such chunks can be found, angelic choice takes over.

We do not allow pattern variables (as in assertions) for these commands: **open** and **close** are *ghost commands*, i.e. commands that aid in verification, but must not alter the runtime semantics of a program. Allowing them to modify variables would be contrary to this objective.

Example 9.9.3. *This example illustrates the possibility of ambiguous matches Sect. 8.2.5 alluded to. If we flip the arguments of the* List *predicate in Example 9.9.2, we can image encountering a situation where the heap contains the chunks* $\mathrm{List}(5, p)$ *and* $\mathrm{List}(5, q)$*, i.e. two chunks expressing the presence of lists of size 5, pointed to by p and q respectively. Modifying the head on the first list (e.g.* $[p] := 3$*) necessitates the manual unfolding of the first chunk:* **open** $\mathrm{List}(5, \_)$*. Two chunks match, an angelic choice is made, resulting in the following two heaps:*

$$\begin{aligned} h_1 = &\quad \{\, p \mapsto v,\, p \mapsto p',\, \mathsf{mb}(p, 2),\, \mathrm{List}(4, p'),\, \mathrm{List}(5, q) \,\} \\ h_2 = &\quad \{\, q \mapsto v,\, q \mapsto q',\, \mathsf{mb}(q, 2),\, \mathrm{List}(4, q'),\, \mathrm{List}(5, p) \,\} \end{aligned}$$

for some $p, p', q, q' \in \mathbb{N}$*.*

Executing $[p] := 3$ *requires the presence of a* $p \mapsto v$ *chunk on the current heap. The heap* h_1 *satisfies this condition, and execution can proceed in the first execution path. The second path fails, but since the forking was angelic, it has no consequences with respect to verification.*

To recapitulate: in order to avoid execution trees from attaining infinite depths, we modularised verification on a per routine basis. For this, we introduced routine contracts, which in turn required us to introduce assertions, user defined predicates and the open/close command pair. We are now prepared to fully formalise the semiconcrete execution.

We start with defining the new state. The only difference between concrete and semiconcrete states lies in the fact that the latter can contain custom chunks.

Definition 9.9.6 (`SemiconcreteExecution.SCStore.t`)**.** *A* semiconcrete store *is defined in the same way as a concrete store (Def. 9.8.1, page 143):*

$$SCStore \equiv Id \to \mathbb{N}$$

Definition 9.9.7 (`SemiconcreteExecution.SCChunk.t`)**.** Semiconcrete chunks *are applications of predicates to integer arguments.*

$$\alpha \in SCChunk = \{\, p(v, v') \mid p \in PredicateName,\ v, v' \in \mathbb{N} \,\}$$

Definition 9.9.8 (`SemiconcreteExecution.SCHeap.t`). *A* semiconcrete heap *is a multiset of semiconcrete chunks.*

$$SCHeap = SCChunk \to \mathbb{N}$$

Note that the set of semiconcrete heaps is a superset of the concrete heaps.

Definition 9.9.9 (`SemiconcreteExecution.semiconcrete_state`). *A* semiconcrete execution state $\langle s, h \rangle_{\mathrm{sc}}$ *consists of a semiconcrete store s and a semiconcrete heap h.*

$$\sigma \in SCState = SCStore \times SCHeap$$

We denote a semiconcrete state consisting of a store s and a heap h with $\langle s, h \rangle_{\mathrm{sc}}$.

Having defined the state, we can proceed with the operators. The operators defined for the concrete execution are usable for the semiconcrete execution, so there is no need to redefine them: we only have to update their type so that they accept semiconcrete states.

Definition 9.9.10 (`SemiconcreteExecution.sc_operator`). *A semiconcrete operator is an operator working on semiconcrete states.*

$$SCOperator_A \equiv SCOperator_A^{SCState \to SCState}$$

Operators to deal with assertions are necessary. consume-assertion(a) checks that the current state satisfies[9] a and removes all involved chunks in the process. produce-assertion(a) does the opposite: it transforms the current state into one that satisfies the assertion by producing the necessary chunks and ensuring that the specified conditions are met.

Definition 9.9.11 (`SemiconcreteExecution.consume_assertion`). *Consuming an assertion performs the following steps:*

- *It makes sure the current state satisfies the necessary logical conditions (described by boolean expressions) in the assertion. If not, failure ensues.*
- *The current heap must contain chunks that match the assertion's. If not, failure ensues. If a match is found, the second argument is bound to the given variable.*
- *A conditional is dealt with by producing the assertion in the appropriate branch. The other branch is blocked by the assumption.*

[9]More precisely, satisfaction of an assertion a only requires that there is a *part* of the heap which is described by a. For example, both semiconcrete states $\langle s_0, \{1 \mapsto 1\} \rangle_{\mathrm{sc}}$ and $\langle s_0[\mathrm{x} := 5], \{1 \mapsto 1, 2 \mapsto 2\} \rangle_{\mathrm{sc}}$ satisfy the assertion $1 \mapsto ?v$, but only the latter satisfies $1 \mapsto ?v \star 2 \mapsto ?w$.

$$\begin{array}{rcl}
\textsf{consume-assertion} & : & \mathit{Assertion} \to \mathit{SCOperator}_{\text{unit}} \\
\hline
\textsf{consume-assertion}(b) & = & \textbf{do } \textsf{assert-bexpr}(b) \\
\\
\textsf{consume-assertion}(p(e, ?x)) & = & \textbf{do } n_1 \leftarrow \textsf{evaluate}(e) \\
& & \quad\ p(n_1, n_2) \leftarrow \textsf{find-chunk}(p, n_1) \\
& & \quad\ \textsf{consume-chunk}(p(n_1, n_2)) \\
& & \quad\ \textsf{update-store}(x, n_2) \\
\\
\textsf{consume-assertion}(a_1 \star a_2) & = & \textbf{do } \textsf{consume-assertion}(a_1) \\
& & \quad\ \textsf{consume-assertion}(a_2)
\end{array}$$

$$\begin{array}{l}
\textsf{consume-assertion}(\textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2) = \\
\quad \textbf{do } \ op \leftarrow \textsf{pick}_{\text{d}}\left(\left\{ \begin{array}{c} \textsf{assume-bexpr}(b) \gg \textsf{consume-assertion}(a_1) \\ \textsf{assume-bexpr}(\neg_{\text{b}} b) \gg \textsf{consume-assertion}(a_2) \end{array} \right\}\right) \\
\qquad\quad op
\end{array}$$

Definition 9.9.12 (`SemiconcreteExecution.produce_assertion`). *The following steps are performed by the* produce-assertion *operator:*

- *Conditions are assumed, i.e. if the assertion requires a certain condition to be met, all execution paths which do not satisfy it are blocked.*
- *The heap is populated with the chunks mentioned in the assertion. The first argument is specified by the assertion. The second argument is unspecified and hence is picked demonically, after which it is bound to the given variable.*
- *Conditionals are dealt with in a straightforward manner.*

$$\begin{array}{rcl}
\textsf{produce-assertion} & : & \mathit{Assertion} \to \mathit{SCOperator}_{\text{unit}} \\
\hline
\textsf{produce-assertion}(b) & = & \textbf{do } \textsf{assume-bexpr}(b) \\
\\
\textsf{produce-assertion}(p(e, ?x)) & = & \textbf{do } n_1 \leftarrow \textsf{evaluate}(e) \\
& & \quad\ n_2 \leftarrow \textsf{pick}_{\text{d}}(\mathbb{N}) \\
& & \quad\ \textsf{produce-chunk}(p(n_1, n_2)) \\
& & \quad\ \textsf{update-store}(x, n_2) \\
\\
\textsf{produce-assertion}(a_1 \ \star \ a_2) & = & \textbf{do } \textsf{produce-assertion}(a_1) \\
& & \quad\ \textsf{produce-assertion}(a_2)
\end{array}$$

$$\begin{array}{l}
\textsf{produce-assertion}(\textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2) = \\
\quad \textbf{do } \ op \leftarrow \textsf{pick}_{\text{d}}\left(\left\{ \begin{array}{ccc} \textsf{assume-bexpr}(b) & \gg & \textsf{produce-assertion}(a_1) \\ \textsf{assume-bexpr}(\neg_{\text{b}} b) & \gg & \textsf{produce-assertion}(a_2) \end{array} \right\}\right) \\
\qquad\quad op
\end{array}$$

Example 9.9.4. *Consider the following assertion:*

$$\mathrm{A}(\mathrm{x}, ?\mathrm{y}) \star (\textbf{if } \mathrm{y} =_\mathrm{b} 0 \textbf{ then } \mathrm{B}(\mathrm{y}, ?\mathrm{z}) \textbf{ else } \mathrm{C}(\mathrm{y}, ?\mathrm{z})) \star \mathrm{z} =_\mathrm{b} 3$$

Producing it in the state $\langle s_0[\mathrm{x} := 5], \emptyset\rangle_{\mathrm{sc}}$*, we get the following result:*

$$\begin{gathered} [\![\langle s_0[\mathrm{x} := 5][\mathrm{y} := 0][\mathrm{z} := 3], \{\mathrm{A}(5, 0), \mathrm{B}(0, 3)\}\rangle_{\mathrm{sc}}]\!] \otimes \\ \bigotimes_{v \in \mathbb{N}_0} [\![\langle s_0[\mathrm{x} := 5][\mathrm{y} := v][\mathrm{z} := 3], \{\mathrm{A}(5, v), \mathrm{C}(v, 3)\}\rangle_{\mathrm{sc}}]\!] \end{gathered}$$

Notice the equality $\mathrm{z} =_\mathrm{b} 3$ *must come last: if it had appeared before the conditional,* z *would still be bound to* 0*, and production would have blocked.*

Definition 9.9.13 (`SemiconcreteExecution.semiconcrete_execution`). *The operator* sc-execute *formalises the semiconcrete execution. Fig. 9.7 shows its full definition.*

Now that we have fully formalised the semiconcrete execution, we need to redefine what it means to verify a program using the semiconcrete execution.

Definition 9.9.14 (`SemiconcreteExecution.leak_check`). leak-check *fails if the heap is not empty.*

$$\begin{aligned} \text{leak-check} = \;\; & \textbf{do } \; h \leftarrow \text{current-heap} \\ & \phantom{\textbf{do }} \;\; \text{assert } h = \emptyset \end{aligned}$$

To define program validity, we first focus on routine validity. Informally, a routine is valid if when starting in an arbitrary state satisfying the precondition, execution of the body ends up in a state satisfying the postcondition.

Definition 9.9.15 (`SemiconcreteExecution.valid_routine`). *A routine r with the following definition*

$$\textbf{routine } r(x) \textbf{ requires } a \textbf{ ensures } a' \;=\; c$$

is valid iff

$$\text{valid-routine}_{\mathrm{sc}}(r) \equiv \left(\begin{array}{ll} \textbf{do} & v \leftarrow \text{pick}_\mathrm{d}(\mathbb{N}) \\ & \text{update-store}(x, v) \\ & \text{produce-assertion}(a) \\ & \text{with-store}\left(\begin{array}{l} s_0[x := v], \\ \text{sc-execute}(c) \end{array} \right) \\ & \text{consume-assertion}(a') \\ & \text{leak-check} \end{array} \right) (\langle s_0, \emptyset\rangle_{\mathrm{sc}}) \not\Leftrightarrow \bot$$

In words, checking routine validity consists of the following steps:

$$
\begin{array}{rll}
\textsf{sc-exec}(x := e) = & \textbf{do} & v \leftarrow \textsf{evaluate}(e); \textsf{update-store}(x, v) \\[1ex]
\textsf{sc-exec}(x := [e]) = & \textbf{do} & \ell \leftarrow \textsf{evaluate}(e); v \leftarrow \textsf{read-cell}(\ell) \\
& & \textsf{update-store}(x, v) \\[1ex]
\textsf{sc-exec}([e] := e') = & \textbf{do} & \ell \leftarrow \textsf{evaluate}(e); v \leftarrow \textsf{evaluate}(e') \\
& & \textsf{write-cell}(\ell, v) \\[1ex]
\textsf{sc-exec}(c; c') = & \textbf{do} & \textsf{sc-exec}(c); \textsf{sc-exec}(c') \\[1ex]
\textsf{sc-exec}(\textbf{if } b \textbf{ then } c \textbf{ else } c') = & \textbf{do} & op \leftarrow \textsf{pick}_{\textsf{d}} \left(\begin{Bmatrix} \textsf{A}(b) \gg \textsf{sc-exec}(c) \\ \textsf{A}(\neg_{\textsf{b}} b) \gg \textsf{sc-exec}(c') \end{Bmatrix} \right); op \\[1ex]
\textsf{sc-exec}(r(e)) = & \textbf{do} & v \leftarrow \textsf{evaluate}(e) \\
& & \textsf{with-store}(s_0[x := v], \textsf{C}(a) \gg \textsf{P}(a')) \\
& \multicolumn{2}{l}{\text{with } \textbf{routine } r(x) \textbf{ requires } a \textbf{ ensures } a'} \\[1ex]
\textsf{sc-exec}(x := \textbf{malloc}(n)) = & \textbf{do} & \ell \leftarrow \textsf{allocate}(n) \\
& & \textsf{produce-chunk}(\textsf{mb}(\ell, n)) \\
& & \textsf{update-store}(x, \ell) \\[1ex]
\textsf{sc-exec}(\textbf{free}(e)) = & \textbf{do} & \ell \leftarrow \textsf{evaluate}(x); n \leftarrow \textsf{block-size}(\ell) \\
& & \textsf{consume-chunk}(\textsf{mb}(\ell, n)) \\
& & \textsf{consume-cells}(\ell, n) \\[1ex]
\textsf{sc-exec}(\textbf{open } p(e, \_)) = & \textbf{do} & \ell \leftarrow \textsf{evaluate}(e) \\
& & p(\ell, v) \leftarrow \textsf{find-chunk}(p, \ell) \\
& & \textsf{consume-chunk}(p(\ell, v)) \\
& & \textsf{with-store}(s_0[x := \ell][y := v], \textsf{P}(a)) \\
& \multicolumn{2}{l}{\text{where } \textbf{predicate } p(x, y) = a} \\[1ex]
\textsf{sc-exec}(\textbf{close } p(e, e')) = & \textbf{do} & \ell \leftarrow \textsf{evaluate}(e); v \leftarrow \textsf{evaluate}(e') \\
& & \textsf{with-store}(s_0[x := \ell][y := v], \textsf{C}(a)) \\
& & \textsf{produce-chunk}(p(\ell, v)) \\
& \multicolumn{2}{l}{\text{where } \textbf{predicate } p(x, y) = a} \\[1ex]
\textsf{sc-exec}(\textbf{skip}) = & \textbf{do} & \textsf{return } \square
\end{array}
$$

$\textsf{P} = \textsf{produce-assertion} \quad \textsf{C} = \textsf{consume-assertion} \quad \textsf{A} = \textsf{assume-bexpr}$

Figure 9.7: Semiconcrete Execution

1. We demonically pick an arbitrary value for the argument and bind it to the appropriate variable.
2. We produce the assertion, i.e. we set up a state which conforms to the precondition.
3. The routine's body is executed in a separate store: the precondition may contain variable patterns, leading variables to be bound to values. These must not be available to the routine body.
4. Consuming the assertion corresponds to checking that the postcondition is satisfied. Note that this happens in the same store as the one the operator produce-assertion(a) operated upon, meaning all bindings introduced by the precondition are available to the postcondition.
5. The postcondition should describe the entire heap. Recall that the operator consume-assertion(a') also removes all chunks mentioned in the assertion. Leftover chunks indicates the presence of a memory leak.

Definition 9.9.16 (`SemiconcreteExecution.valid_program`). *Using the semiconcrete execution, a program is valid if every routine it contains is valid:*

$$\text{valid-program}_{\text{sc}}(P) = \forall\, r \in \text{routines}(P).\ \text{valid-routine}_{\text{sc}}(r)$$

9.9.2 Relation with Concrete Execution

In this section we examine how the semiconcrete execution relates to the concrete execution. Earlier, we explained that verification consists of checking that the concrete execution of a program does not fail. Our goal is to be able to find out whether this is the case without actually performing the concrete execution by instead relying on the semiconcrete execution.

More specifically, we wish that the following relation holds: if the concrete execution of a program were to fail, the semiconcrete execution of the same program should also fail. Turned around, successful semiconcrete execution must imply successful concrete execution. In formal terms, we wish the following proposition to be true:

$$\forall\, P.\ \text{valid-program}_{\text{sc}}(P) \Rightarrow \text{valid-program}_{\text{c}}(P)$$

If this is the case, we say the semiconcrete execution is *sound*. Turning the implication arrow around

$$\forall\, P.\ \text{valid-program}_{\text{c}}(P) \Rightarrow \text{valid-program}_{\text{sc}}(P)$$

is known as *completeness*. Our claim is that the semiconcrete execution is sound, but we make no guarantees regarding completeness. In other words, we do not allow false positives (i.e. a faulty program being recognised as correct), but accept the possibility of false negatives (i.e. rejection of a correct program).

One first hurdle in comparing both executions is that they operate on different languages, namely the concrete execution uses SIL, while the semiconcrete uses it extended variant, SIL^{++}. The latter provides extra commands, i.e. **open** and **close**. Fortunately, since these commands have no bearing on the runtime semantics of a program, we can leave them out for the purposes of concrete execution.

Definition 9.9.17 (`SILPP.translate_command`). *Given a semiconcrete execution command, we translate it to a concrete execution command as follows:*

$$\begin{array}{lcl}
\lfloor x := e \rfloor & = & x := e \\
\lfloor e := [e'] \rfloor & = & e := [e'] \\
\lfloor [e] := e' \rfloor & = & [e] := e' \\
\lfloor c; c' \rfloor & = & \lfloor c \rfloor ; \lfloor c' \rfloor \\
\lfloor \textbf{if } b \textbf{ then } c \textbf{ else } c \rfloor & = & \textbf{if } b \textbf{ then } \lfloor c \rfloor \textbf{ else } \lfloor c' \rfloor \\
\lfloor r(e) \rfloor & = & r(e) \\
\lfloor x := \textbf{malloc}(n) \rfloor & = & x := \textbf{malloc}(n) \\
\lfloor \textbf{free}(e) \rfloor & = & \textbf{free}(e) \\
\lfloor \textbf{open } p(e, \_) \rfloor & = & \textbf{skip} \\
\lfloor \textbf{close } p(e, e') \rfloor & = & \textbf{skip} \\
\lfloor \textbf{skip} \rfloor & = & \textbf{skip}
\end{array}$$

In order to prove soundness, we prove that the concrete and semiconcrete execution are synchronised in some way, that the results are "equivalent". A first approach would be to expect the following to be true (expressed intuitively):

$$\text{sc-execute}(c) \overset{\cdot}{\iff} \text{c-execute}(\lfloor c \rfloor)$$

This represents both soundness and completeness. We can weaken our expectation to only soundness:

$$\text{sc-execute}(c) \overset{\cdot}{\Rightarrow} \text{c-execute}(\lfloor c \rfloor)$$

A consequence of this proposition's validity is that if the semiconcrete execution succeeds, so must the concrete execution (Lemma 9.3.9), which is exactly what we need. However, the types don't match: we cannot directly compare a result involving semiconcrete states with one involving concrete states.

Semiconcrete and concrete states differ in the fact that the former can contain custom chunks. A custom chunk can repeatedly be unfolded until it exclusively contains concrete chunks, i.e. $\mapsto$ and mb chunks. Thus, a single semiconcrete state can be viewed as representing a multitude of concrete states.

Example 9.9.5. *This example revisits the* List *predicate:*

$$\begin{array}{ll}\textbf{predicate}\ \mathrm{List}(p,n) = & \textbf{if}\ p =_{\mathrm{b}} 0 \\ & \textbf{then}\ n =_{\mathrm{b}} 0 \\ & \textbf{else}\ \ p \mapsto ?v \star \\ & \qquad (p +_{\mathrm{e}} 1) \mapsto ?q \star \\ & \qquad \mathsf{mb}(p,2) \star \\ & \qquad \mathrm{List}(q, ?m) \star \\ & \qquad n =_{\mathrm{b}} m +_{\mathrm{e}} 1\end{array}$$

The single semiconcrete state

$$\langle s_0[\mathrm{p} := 3], \{\mathrm{List}(3,2)\}\rangle_{\mathrm{sc}}$$

represents all *states where the variable* p *points to a list of length 2 whose first node starts at memory address* 3*. Repeatedly unfolding the* List *chunk yields the following semiconcrete states:*

$$\langle s_0[\mathrm{p} := 3], \{3 \mapsto \ell, 4 \mapsto v, \mathsf{mb}(3,2), \ell \mapsto 0, \ell+1 \mapsto v', \mathsf{mb}(\ell,2)\}\rangle_{\mathrm{sc}}$$

for all $\ell, v, v' \in \mathbb{N}$. Since all these semiconcrete states only contain concrete chunks, they are also valid concrete states. We will show that semiconcretely executing a semiconcrete state corresponds to concretely executing its unfolded forms.

Definition 9.9.18. $\sigma_{\mathrm{sc}} \Vdash a$ *is valid iff the assertion a fully describes the concrete state σ_{sc}, i.e. the assertion mentions all chunks in the state's heap.*

$$\langle s,h\rangle_{\mathrm{sc}} \Vdash a \quad \Leftrightarrow \quad \begin{pmatrix}\textbf{do} & \mathsf{consume\text{-}assertion}(a) \\ & \mathsf{leak\text{-}check}\end{pmatrix}(\langle s,h\rangle_{\mathrm{sc}}) \not\Leftarrow\!\!\!\Rightarrow \bot$$

Definition 9.9.19 (heap refinement)**.** *The* heap refinement relation *relates a concrete heap with a semiconcrete heaps according to the following rules:*

$$\frac{h \in \mathit{CHeap}}{h \trianglelefteq h}\ \trianglelefteq\text{-}\textsc{reflexivity}$$

$$\frac{h_{\mathrm{c}} \trianglelefteq h_{\mathrm{sc}} \qquad h'_{\mathrm{c}} \trianglelefteq h'_{\mathrm{sc}}}{h_{\mathrm{c}} \uplus h'_{\mathrm{c}} \trianglelefteq h_{\mathrm{sc}} \uplus h'_{\mathrm{sc}}}\ \trianglelefteq\text{-}\textsc{union}$$

$$\frac{h_{\mathrm{c}} \trianglelefteq h_{\mathrm{sc}} \quad \textbf{predicate}\ p(x,x') = a \quad \langle s_0[x := \ell][x' := v], h_{\mathrm{sc}}\rangle_{\mathrm{sc}} \Vdash a}{h_{\mathrm{c}} \trianglelefteq \{\ p(\ell,v)\ \}}\ \trianglelefteq\text{-}\textsc{predicate}$$

Heap refinement thus embodies the notion of "full unfolding" of a semiconcrete heap.

Example 9.9.6. *The unfolding from Example 9.9.5 can be reformulated formally as*

$$\forall\, \ell, v, v' \in \mathbb{N}.\ \langle s_0[\mathsf{p} := 3], \left\{\begin{array}{c} 3 \mapsto \ell, \\ 4 \mapsto v, \\ \mathsf{mb}(3,2), \\ \ell \mapsto 0, \\ \ell + 1 \mapsto v', \\ \mathsf{mb}(\ell, 2) \end{array}\right\}\rangle_{\mathrm{sc}} \trianglelefteq \langle s_0[\mathsf{p} := 3], \{\mathrm{List}(3,2)\}\rangle_{\mathrm{sc}}$$

The following operator is pivotal in proving the soundness of the semiconcrete execution.

Definition 9.9.20 (refinement operator)**.** *The* refinement operator κ *is defined as*

$$\begin{array}{lll} \kappa & : & \mathit{Operator}^{\mathit{SCState} \to \mathit{CState}}_{\mathrm{unit}} \\ \hline \kappa & = & \textbf{do}\ \langle s, h_{\mathrm{sc}}\rangle_{\mathrm{sc}} \leftarrow \mathsf{current\text{-}state} \\ & & \quad h_{\mathrm{c}} \leftarrow \mathsf{pick}_{\mathrm{d}}(\mathit{SCHeap}) \\ & & \quad \mathsf{assume}\ h_{\mathrm{c}} \trianglelefteq h_{\mathrm{sc}} \\ & & \quad \mathsf{set\text{-}current\text{-}state}\ \langle s, h_{\mathrm{c}}\rangle_{\mathrm{c}} \end{array}$$

This operator crosses execution boundaries: it takes a semiconcrete state as input and returns a result involving concrete states.

Example 9.9.7. *Example 9.9.6 can be rephrased as*

$$\kappa(\langle s_0[\mathsf{p} := 3], \{\mathrm{List}(3,2)\}\rangle_{\mathrm{sc}}) \Longleftrightarrow \bigoplus_{\ell \in \mathbb{N}} \bigoplus_{v \in \mathbb{N}} \bigoplus_{v' \in \mathbb{N}} [\![\langle s_0[\mathsf{p} := 3], \left\{\begin{array}{c} 3 \mapsto \ell, \\ 4 \mapsto v, \\ \mathsf{mb}(3,2), \\ \ell \mapsto 0, \\ \ell + 1 \mapsto v', \\ \mathsf{mb}(\ell, 2) \end{array}\right\}\rangle_{\mathrm{sc}}]\!]$$

Using this operator, we can relate semiconcrete with concrete execution in a well typed manner:

$$\mathsf{sc\text{-}execute}(c) \gg \kappa \quad \dot{\Rrightarrow} \quad \kappa \gg \mathsf{c\text{-}execute}(\lfloor c \rfloor)$$

This expresses the fact that semiconcretely executing a command in a semiconcrete state σ_{sc} is a sound approximation to concretely executing the translated command in a refinement of that state $\sigma_{\mathrm{c}} \trianglelefteq \sigma_{\mathrm{sc}}$. Before we are able to prove this, we need to turn our attention to another modification introduced by the semiconcrete execution that we have been ignoring until now: modularisation. We set out to prove its validity.

Definition 9.9.21 (local operator)**.** *An operator op is said to be* local *iff*

$$\mathbf{local}(op) \equiv \left(\begin{array}{ll}\mathbf{do} & r \leftarrow op \\ & \textsf{produce-chunk}(\alpha) \\ & \textsf{return } r\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & \textsf{produce-chunk}(\alpha) \\ & op\end{array}\right)$$

The concept of local operators is strongly linked with separation logic. Recall the frame rule discussed in Sect. 8.1.2 (page 80). Explained briefly, it states that one can add heap chunks to the state without changing the behaviour of a command. Note that Def. 9.9.21 uses implication ($\dot{\Rrightarrow}$), not equivalence ($\dot{\Longleftrightarrow}$): this mirrors the fact that moving a produce-chunk upwards can make a failing operator succeed and potentially increasing the number of ambiguous matches. We rely on this property for the soundness proof. Hence, we need to prove that our executions obey this frame rule.

Lemma 9.9.1. *Binding two local operators yields a new local operator: given*

$$\begin{array}{rcl} op & : & Operator_A^{\mathcal{S}_1 \to \mathcal{S}_2} \\ f & : & A \to Operator_B^{\mathcal{S}_2 \to \mathcal{S}_3} \\ & & \mathbf{local}(op) \\ & & \forall\, a : A.\ \mathbf{local}(f(a)) \end{array}$$

then

$$\mathbf{local}(op \gg\!= f)$$

Proof. The lhs of the proof goal is equivalent with

$$\left(\begin{array}{ll}\mathbf{do} & r \leftarrow op \gg\!= f \\ & \textsf{produce-chunk}(\alpha) \\ & \textsf{return } r\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & \textsf{produce-chunk}(\alpha) \\ & op \gg\!= f\end{array}\right)$$

It is possible to rewrite the lhs as

$$lhs \dot{\Longleftrightarrow} \begin{array}{ll}\mathbf{do} & r' \leftarrow op \\ & r \leftarrow f(r') \\ & \textsf{produce-chunk}(\alpha) \\ & \textsf{return } r\end{array}$$

Using the locality of $f(r')$ and op:

$$\ldots \dot{\Rrightarrow} \begin{array}{ll}\mathbf{do} & r' \leftarrow op \\ & \textsf{produce-chunk}(\alpha) \\ & f(r')\end{array} \qquad \dot{\Rrightarrow} \begin{array}{ll}\mathbf{do} & \textsf{produce-chunk}(\alpha) \\ & r' \leftarrow op \\ & f(r')\end{array}$$

which can be folded back into

$$\ldots \dot{\Rrightarrow} \ \mathbf{do} \ \begin{array}{l} \text{produce-chunk}(\alpha) \\ op \gg\!= f \end{array}$$

□

Lemma 9.9.2. *The execution operators are local operators.*

Proof. From the locality of auxiliary operators follows the locality of the high level execution operators. Proving the locality of every local operator is tedious and uninteresting. Locality proofs depend mostly on Lemma 9.9.1 and Lemma 9.7.1. □

The following lemmas show how assertion consumption and production complement each other. For example, Lemma 9.9.3 proves that assertion production followed by consumption (executed locally in the same store) cannot fail and corresponds to a no-op except for the fact that blocking is allowed, such as is the case for the assertion "false".

Lemma 9.9.3 (Produce-consume cancellation law)**.**

$$\text{nop} \quad \dot{\Rrightarrow} \quad \mathbf{do} \ \begin{array}{l} \text{with-store}(s, \text{produce-assertion}(a)) \\ \text{with-store}(s, \text{consume-assertion}(a)) \end{array}$$

Proof. We introduce the following operator:

$$\text{aux}(s_{\text{in}}, op) = \ \mathbf{do} \ \begin{array}{l} s_{\text{orig}} \leftarrow \text{current-store} \\ \text{set-store}(s_{\text{in}}) \\ op \\ s_{\text{out}} \leftarrow \text{current-store} \\ \text{set-store}(s_{\text{orig}}) \\ \text{return } s_{\text{out}} \end{array}$$

and the following shorthand notation

$$\begin{array}{rcl} s_{\text{in}} \blacktriangleright op \blacktriangleright s_{\text{out}} & \equiv & s_{\text{out}} \leftarrow \text{aux}(s_{\text{in}}, op) \\ P(a) & \equiv & \text{produce-assertion}(a) \\ C(a) & \equiv & \text{consume-assertion}(a) \end{array}$$

We generalise our goal to

$$\text{nop} \ \dot{\Rrightarrow} \left(\mathbf{do} \ \begin{array}{l} s \blacktriangleright P(a) \blacktriangleright s_1 \\ s \blacktriangleright C(a) \blacktriangleright s_2 \\ \text{assert } s_1 = s_2 \end{array} \right)$$

This entails that the rhs operator must not fail and must either diverge or produce a path which amounts to a no-op, or, put formally, $\forall\ \sigma.\ \{(\square, \sigma)\} \models lhs(\sigma)$. In other words, both store and heap must remain unchanged. It is clear that the store is preserved, as production and consumption of the assertion is done in a local store and assert never alters the store in any way. By structural induction on a:

- $a = b$

$$\textsf{nop} \ \dot{\Rrightarrow} \left(\begin{array}{l} \textbf{do}\ \ s \blacktriangleright \textsf{assume-bexpr}(b) \blacktriangleright s_1 \\ \qquad s \blacktriangleright \textsf{assert-bexpr}(b) \blacktriangleright s_2 \\ \qquad \textsf{assert}\ s_1 = s_2 \end{array} \right)$$

  The rhs will never fail, as all states not satisfying b are filtered away by the time the assertion takes place. Neither assume-bexpr(b) nor assert-bexpr(b) modifies the store, meaning that $s_1 = s_2 = s$, making the last assert succeed.

- $a = p(e, ?x)$

$$\textsf{nop} \ \dot{\Rrightarrow} \left(\begin{array}{l} \textbf{do}\ \ s \blacktriangleright \left(\begin{array}{l} \textbf{do}\ \ n_1 \leftarrow \textsf{evaluate}(e) \\ \qquad n_2 \leftarrow \textsf{pick}_\textsf{d}(\mathbb{N}) \\ \qquad \textsf{produce-chunk}(p(n_1, n_2)) \\ \qquad \textsf{update-store}(x, n_2) \end{array} \right) \blacktriangleright s_1 \\ \qquad s \blacktriangleright \left(\begin{array}{l} \textbf{do}\ \ n'_1 \leftarrow \textsf{evaluate}(e) \\ \qquad p(n'_1, n'_2) \leftarrow \textsf{find-chunk}(p, n'_1) \\ \qquad \textsf{consume-chunk}(p(n'_1, n'_2)) \\ \qquad \textsf{update-store}(x, n'_2) \end{array} \right) \blacktriangleright s_2 \\ \qquad \textsf{assert}\ s_1 = s_2 \end{array} \right)$$

  The result of the production phase is

$$production(\langle s_0, h_0 \rangle_\text{sc}) \iff \bigotimes_{v \in \mathbb{N}} [\![(\square, \langle s_0, h_0 \uplus \{p(n_1, v)\} \rangle_\text{sc})]\!]$$

  and s_1 is bound to $s[x := v]$ in each path. Continuing with the consumption, we get: ($\Rrightarrow$ but not $\iff$ do due possible ambiguous matches)

$$\bigotimes_{v \in \mathbb{N}} [\![\langle s_0, h_0 \rangle_\text{sc}]\!] \Rrightarrow \bigotimes_{v \in \mathbb{N}} consumption(\langle s_0, h_0 \uplus \{p(n_1, v)\} \rangle_\text{sc})$$

  where s_2 is bound to $s[x := v]$ in each path. The assertion succeeds since $s_1 = s_2$. We can collapse the demonic choice into a single result:

$$\bigotimes_{v \in \mathbb{N}} [\![\langle s_0, h_0 \rangle_\text{sc}]\!] \iff [\![\langle s_0, h_0 \rangle_\text{sc}]\!]$$

  It is now trivial to prove the goal.

- $a = \textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2$: the boolean expression b is evaluated twice in the same store, meaning the same branch will be chosen during both production and consumption. The induction hypothesis takes care of the rest.

- $a = a_1 \star a_2$

  $$\mathsf{nop} \Rrightarrow \left(\begin{array}{l} \textbf{do}\ \ s \blacktriangleright P(a_1) \gg P(a_2) \blacktriangleright s_1 \\ \qquad s \blacktriangleright C(a_1) \gg C(a_2) \blacktriangleright s_2 \\ \qquad \mathsf{assert}\ s_1 = s_2 \end{array}\right)$$

  We can split this up in

  $$\mathsf{nop} \Rrightarrow \left(\begin{array}{l} \textbf{do}\ \ s \blacktriangleright P(a_1) \blacktriangleright s_1' \\ \qquad s_1' \blacktriangleright P(a_2) \blacktriangleright s_1 \\ \qquad s \blacktriangleright C(a_1) \blacktriangleright s_2' \\ \qquad s_2' \blacktriangleright C(a_2) \blacktriangleright s_2 \\ \qquad \mathsf{assert}\ s_1 = s_2 \end{array}\right)$$

  We switch the second and third line:

  $$\mathsf{nop} \Rrightarrow \left(\begin{array}{l} \textbf{do}\ \ s \blacktriangleright P(a_1) \blacktriangleright s_1' \\ \qquad s \blacktriangleright C(a_1) \blacktriangleright s_2' \\ \qquad s_1' \blacktriangleright P(a_2) \blacktriangleright s_1 \\ \qquad s_2' \blacktriangleright C(a_2) \blacktriangleright s_2 \\ \qquad \mathsf{assert}\ s_1 = s_2 \end{array}\right) \Rrightarrow \left(\begin{array}{l} \textbf{do}\ \ s \blacktriangleright P(a_1) \blacktriangleright s_1' \\ \qquad s_1' \blacktriangleright P(a_2) \blacktriangleright s_1 \\ \qquad s \blacktriangleright C(a_1) \blacktriangleright s_2' \\ \qquad s_2' \blacktriangleright C(a_2) \blacktriangleright s_2 \\ \qquad \mathsf{assert}\ s_1 = s_2 \end{array}\right)$$

  This switch is permitted and the result is "stricter": $P(a_2)$ exclusively produces new chunks, making use of the locality property of $C(a_1)$ possible. From the induction hypothesis we know that $s_1' = s_2'$. It also allows us to replace the first two lines with nop.

  $$\mathsf{nop} \Rrightarrow \left(\begin{array}{l} \textbf{do}\ \ s_1' \blacktriangleright P(a_2) \blacktriangleright s_1 \\ \qquad s_1' \blacktriangleright C(a_2) \blacktriangleright s_2 \\ \qquad \mathsf{assert}\ s_1 = s_2 \end{array}\right)$$

  We can again apply the induction hypothesis, finishing the proof.

□

The following lemma is a generalised form of Lemma 9.9.5 and Lemma 9.9.6, provable by induction.

Lemma 9.9.4. *For any semiconcrete store s,*

$$\left(\begin{array}{l} \textbf{do}\ \ s_1 \leftarrow \mathsf{with\text{-}store}(s, \mathsf{consume\text{-}assertion}(a) \ggg \mathsf{current\text{-}store}) \\ \qquad s_2 \leftarrow \mathsf{with\text{-}store}(s, \mathsf{produce\text{-}assertion}(a) \ggg \mathsf{current\text{-}store}) \\ \qquad \mathsf{assume}(s_1 = s_2) \end{array}\right) \Rrightarrow \mathsf{nop}$$

Proof. We reuse the notation introduced in the proof of Lemma 9.9.3. The proof goal is rewritten as

$$\left(\begin{array}{l}\textbf{do}\ \ s \blacktriangleright C(a) \blacktriangleright s_1 \\ \qquad s \blacktriangleright P(a) \blacktriangleright s_2 \\ \qquad \textsf{assume}\ s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

This allows failure and demands that each angelic choice contains a path equivalent with nop: $\forall\ \sigma, S.\ S \models lhs(\sigma) \Rightarrow (\square, \sigma) \in S$. Note that blocking is not allowed; in other words, we require that $s_1 = s_2$. By structural induction on a,

- $a = b$

  $$\left(\begin{array}{l}\textbf{do}\ \ s \blacktriangleright \textsf{assert-bexpr}(b) \blacktriangleright s_1 \\ \qquad s \blacktriangleright \textsf{assume-bexpr}(b) \blacktriangleright s_2 \\ \qquad \textsf{assume}\ s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

  Failure is allowed, but not nontermination, hence we must show that assume-bexpr does not block. This is the case: assert-bexpr takes care of removing all paths where s does not satisfy b. Neither assert-bexpr nor assume-bexpr change the store, meaning $s_1 = s_2$, preventing the assumption on the last line to block.

- $a =$ **if** b **then** a_1 **else** a_2: b is evaluated in the same store s, meaning the same branch will be picked. Induction hypothesis takes care of the rest.

- $a = p(\ell, v)$

  $$\left(\begin{array}{l}\textbf{do}\ \ s \blacktriangleright \left(\begin{array}{l}\textbf{do}\ \ n_1 \leftarrow \textsf{evaluate}(e) \\ \qquad p(n_1, n_2) \leftarrow \textsf{find-chunk}(p, n_1) \\ \qquad \textsf{consume-chunk}(p(n_1, n_2)) \\ \qquad \textsf{update-store}(x, n_2)\end{array}\right) \blacktriangleright s_1 \\ \qquad s \blacktriangleright \left(\begin{array}{l}\textbf{do}\ \ n_1 \leftarrow \textsf{evaluate}(e) \\ \qquad n_2 \leftarrow \textsf{pick}_\textsf{d}(\mathbb{N}) \\ \qquad \textsf{produce-chunk}(p(n_1, n_2)) \\ \qquad \textsf{update-store}(x, n_2)\end{array}\right) \blacktriangleright s_2 \\ \qquad \textsf{assume}\ s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

  We can disregard cases that fail, meaning that we can assume the heap contains a chunk $p(n_1, n_2)$ where n_1 is equal to the evaluation of e in the initial store. In case of ambiguous matches, we can handle each match analogously. The result of the consumption phase is (assuming consumption matches $p(n_1, n_2)$):

  $$consumption_{p(n_1,n_2)}(\langle s_0, h_0 \uplus \{p(n_1, n_2)\}\rangle_{\text{sc}}) \iff [\![(\square, \langle s_0, h_0\rangle_{\text{sc}})]\!]$$

and s_1 is bound to $s[x := n_2]$. The production phase results in

$$\mathit{production}(\langle s_0, h_0\rangle_{\text{sc}}) \iff \bigotimes_{v\in\mathbb{N}} [\![(\square, \langle s_0, h_0 \uplus \{p(n_1, v)\}\rangle_{\text{sc}})]\!]$$

and s_2 is bound to $s[x := v]$ on each path. The assume blocks every path where $s_1 \neq s_2$. We know there is exactly one path that will remain, making the end result

$$\mathit{lhs}(\langle s_0, h_0\rangle_{\text{sc}}) \iff [\![(\square, \langle s_0, h_0 \uplus \{p(n_1, n_2)\}\rangle_{\text{sc}})]\!]$$

Proving the goal is now trivial.

- $a = a_1 \star a_2$

$$\left(\begin{array}{ll}\textbf{do} & s \blacktriangleright C(a_1) \gg C(a_2) \blacktriangleright s_1 \\ & s \blacktriangleright P(a_1) \gg P(a_2) \blacktriangleright s_2 \\ & \textsf{assume } s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

We rewrite this as

$$\left(\begin{array}{ll}\textbf{do} & s \blacktriangleright C(a_1) \blacktriangleright s_1' \\ & s_1' \blacktriangleright C(a_2) \blacktriangleright s_1 \\ & s \blacktriangleright P(a_1) \blacktriangleright s_2' \\ & s_2' \blacktriangleright P(a_2) \blacktriangleright s_2 \\ & \textsf{assume } s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

We switch the second and third lines:

$$\left(\begin{array}{ll}\textbf{do} & s \blacktriangleright C(a_1) \blacktriangleright s_1' \\ & s_1' \blacktriangleright C(a_2) \blacktriangleright s_1 \\ & s \blacktriangleright P(a_1) \blacktriangleright s_2' \\ & s_2' \blacktriangleright P(a_2) \blacktriangleright s_2 \\ & \textsf{assume } s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\textbf{do} & s \blacktriangleright C(a_1) \blacktriangleright s_1' \\ & s \blacktriangleright P(a_1) \blacktriangleright s_2' \\ & s_1' \blacktriangleright C(a_2) \blacktriangleright s_1 \\ & s_2' \blacktriangleright P(a_2) \blacktriangleright s_2 \\ & \textsf{assume } s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

The induction hypothesis tells us that

$$\left(\begin{array}{ll}\textbf{do} & s \blacktriangleright C(a_1) \blacktriangleright s_1' \\ & s \blacktriangleright P(a_1) \blacktriangleright s_2' \\ & \textsf{assume } s_1' = s_2'\end{array}\right) \dot{\Rrightarrow} \textsf{nop}$$

meaning that $s_1' = s_2'$.

$$\left(\begin{array}{ll}\textbf{do} & s \blacktriangleright C(a_1) \blacktriangleright s_1' \\ & s \blacktriangleright P(a_1) \blacktriangleright s_1' \\ & s_1' \blacktriangleright C(a_2) \blacktriangleright s_1 \\ & s_1' \blacktriangleright P(a_2) \blacktriangleright s_2 \\ & \textsf{assume } s_1 = s_2\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\textbf{do} & s_1' \blacktriangleright C(a_2) \blacktriangleright s_1 \\ & s_1' \blacktriangleright P(a_2) \blacktriangleright s_2 \\ & \textsf{assume } s_1 = s_2\end{array}\right)$$

Applying the induction hypothesis a second time finishes the proof.

□

Whereas Lemma 9.9.3 described assertion production followed by consumption, the following lemma deals with the reverse situation. Contrary to the former, this lemma allows failure but prohibits blocking.

Lemma 9.9.5 (Consume-produce cancellation law)**.** *For any semiconcrete store s,*

$$\left(\begin{array}{ll}\textbf{do} & \textsf{with-store}(s, \textsf{consume-assertion}(a)) \\ & \textsf{with-store}(s, \textsf{produce-assertion}(a))\end{array}\right) \mathrel{\dot{\Rrightarrow}} \textsf{nop}$$

Proof. Follows from Lemma 9.9.4. □

The following lemma can probably use some extra explanation. Intuitively, both assertion consumption and production fork execution and thus generate a number of output stores which the lemma names s_1 and s_2, respectively. Our goal is to prove the existence of an execution path for which the consumption and production generate the same store, i.e. $s_1 = s_2$, and where the heap is restored to its original state, i.e. where production has undone the consumption's heap modifications. This allows us to replace the consumption production sequence by "magically" picking the right store, represented by an angelic choice over the set of all stores. All other execution paths can be ignored.

Lemma 9.9.6. *For the semiconcrete execution*

$$\begin{array}{ll}\textbf{do} & s_1 \leftarrow \textsf{with-store}(s, \textsf{consume-assertion}(a) \gg \textsf{current-store}); \\ & s_2 \leftarrow \textsf{with-store}(s, \textsf{produce-assertion}(a) \gg \textsf{current-store}); \\ & \textsf{return}\ (s_1, s_2)\end{array}$$

$$\Downarrow\!\!\Downarrow\cdot$$

$$\begin{array}{ll}\textbf{do} & s \leftarrow \textsf{pick}_\textsf{a}(\mathit{SCStore}); \\ & \textsf{return}\ (s, s)\end{array}$$

Proof. Follows from Lemma 9.9.4. □

Next, we prove that a routine contract "neutralises" the effects of executing its body. This lemma is central in the proving the soundness of verification modularisation (Lemma 9.9.8), i.e. that replacing the execution of a routine body by consuming and producing its precondition and postcondition, respectively, is a valid approximation.

Lemma 9.9.7. *Given a valid routine* **routine** $r(x)$ **requires** a **ensures** $a' = c$,

$$\mathsf{nop} \Rrightarrow \mathsf{with\text{-}store}(s_0[x := v],\ \mathbf{do}\ \begin{array}{l}\mathsf{produce\text{-}assertion}(a);\\ \mathsf{with\text{-}store}(s_0[x := v], \mathsf{sc\text{-}execute}(c));\\ \mathsf{consume\text{-}assertion}(a'))\end{array}$$

Proof. Follows from routine validity (Def. 9.9.15) and the locality of with-store, produce-assertion, consume-assertion and sc-execute. □

Lemma 9.9.8 (Validity of routine call abstraction)**.** *In the semiconcrete world, given a valid routine*

$$\mathbf{routine}\ r(x)\ \mathbf{requires}\ a\ \mathbf{ensures}\ a' = c$$

then

$$\begin{array}{ll} & \mathbf{do}\ v \leftarrow \mathsf{read\text{-}store}(x);\\ & \quad \mathsf{with\text{-}store}(\ s_0[x := v],\\ & \qquad\qquad \mathbf{do}\ \mathsf{consume\text{-}assertion}(a);\\ & \qquad\qquad\quad\ \ \mathsf{produce\text{-}assertion}(a')\ )\\ \Rrightarrow & \mathbf{do}\ v \leftarrow \mathsf{read\text{-}store}(x);\\ & \quad \mathsf{with\text{-}store}(\ s_0[x := v],\\ & \qquad\qquad \mathsf{sc\text{-}execute}(c)\ ) \end{array}$$

Proof. We reuse the same notation as in the proof of Theorem 9.9.3 and abbreviate produce-assertion, consume-assertion and sc-execute to P, C and E, respectively. From Lemma 9.9.7 we know

$$\mathsf{nop} \Rrightarrow \left(\begin{array}{ll}\mathbf{do} & s_\mathrm{i} \blacktriangleright P(a) \blacktriangleright s_\mathrm{p}\\ & s_\mathrm{i} \blacktriangleright E(c) \blacktriangleright \_\\ & s_\mathrm{p} \blacktriangleright C(a') \blacktriangleright \_\end{array}\right) \tag{9.5}$$

with $s_\mathrm{i} = s_0[x := v]$. We can drop the $v \leftarrow$ read-store(x) from both sides of the proof goal. The lhs can be rewritten as

$$\begin{array}{ll}\mathbf{do} & s_\mathrm{i} \blacktriangleright C(a) \blacktriangleright s'\\ & s' \blacktriangleright P(a') \blacktriangleright \_\end{array}$$

We can insert nop in the middle, and from (9.5) we get

$$\ldots \Rrightarrow \left(\begin{array}{ll}\mathbf{do} & s_\mathrm{i} \blacktriangleright C(a) \blacktriangleright s'\\ & \mathsf{nop}\\ & s' \blacktriangleright P(a') \blacktriangleright \_\end{array}\right) \Rrightarrow \left(\begin{array}{ll}\mathbf{do} & s_\mathrm{i} \blacktriangleright C(a) \blacktriangleright s'\\ & s_\mathrm{i} \blacktriangleright P(a) \blacktriangleright s_\mathrm{p}\\ & s_\mathrm{i} \blacktriangleright E(c) \blacktriangleright \_\\ & s_\mathrm{p} \blacktriangleright C(a') \blacktriangleright \_\\ & s' \blacktriangleright P(a') \blacktriangleright \_\end{array}\right)$$

Using Lemma 9.9.6 on the first two lines gives

$$\ldots \stackrel{\cdot}{\Rrightarrow} \left(\begin{array}{ll} \textbf{do} & s \leftarrow \mathsf{pick}_{\mathsf{a}}(\mathit{SCStore}) \\ & s_{\mathrm{i}} \blacktriangleright E(c) \blacktriangleright \_ \\ & s \blacktriangleright C(a') \blacktriangleright \_ \\ & s \blacktriangleright P(a') \blacktriangleright \_ \end{array} \right)$$

Lemma 9.9.5 rids us of the last two lines:

$$\ldots \stackrel{\cdot}{\Rrightarrow} \left(\begin{array}{ll} \textbf{do} & s \leftarrow \mathsf{pick}_{\mathsf{a}}(\mathit{SCStore}) \\ & s_{\mathrm{i}} \blacktriangleright E(c) \blacktriangleright \_ \end{array} \right)$$

The angelic picking of a semiconcrete store can be dropped, and what remains corresponds to the rhs of the proof goal. □

The following lemma expresses the fact that, after heap refinement, there is no difference between producing a custom chunk or producing its components.

Lemma 9.9.9.

$$\left(\begin{array}{ll} \textbf{do} & \mathsf{with\text{-}store}\left(\begin{array}{l} s_0[x := \ell, y := v], \\ \mathsf{produce\text{-}assertion}(a) \end{array} \right) \\ & \kappa \end{array} \right) \stackrel{\cdot}{\Longleftrightarrow} \left(\begin{array}{ll} \textbf{do} & \mathsf{produce\text{-}chunk}(p(\ell, v)) \\ & \kappa \end{array} \right)$$

where

$$\textbf{predicate } p(x, y) = a$$

Proof. We introduce the following shorthand notations:

$$\begin{array}{lcl} P(a[\ell, v]) & \equiv & \mathsf{with\text{-}store}(s_0[x := \ell, y := v], \mathsf{produce\text{-}assertion}(a)) \\ P(p(\ell, v)) & \equiv & \mathsf{produce\text{-}chunk}(p(\ell, v)) \end{array}$$

First we consider only empty initial heaps. We need to show that the same concrete stores are produced at both sides.

- $\Rrightarrow$: we must prove that every concrete heap generated by $P(p(\ell, v)) \gg \kappa$ is also generated by $P(a[\ell, v]) \gg \kappa$. According to $\trianglelefteq$-PREDICATE, this follows from the fact that $P(a[\ell, v])$ generates all heaps h_{sc} such that $\langle s_0[x := \ell][x' := v], h_{\mathrm{sc}} \rangle_{\mathrm{sc}} \Vdash a$, which in turns follows from Def. 9.9.18 and Lemma 9.9.3.
- $\Lleftarrow$: follows from $\trianglelefteq$-PREDICATE.

We can generalise to nonempty heaps by using $\trianglelefteq$-UNION and noting that the behaviour of produce-assertion is entirely independent of the heap in which it is executed. □

We are now finally ready to prove the semiconcrete execution's soundness. The following theorem expresses that the semiconcrete execution is stricter, or harder to satisfy, than the concrete execution.

Theorem 9.9.1. *For any valid program, the semiconcrete execution is a sound approximation to the concrete execution.*

$$\textsf{sc-execute}(c) \gg \kappa \quad \dot{\Rrightarrow} \quad \kappa \gg \textsf{c-execute}(\lfloor c \rfloor)$$

Proof. We unfold c-execute:

$$\textsf{sc-execute}(c) \gg \kappa \quad \dot{\Rrightarrow} \quad \kappa \gg \bigotimes_{n \in \mathbb{N}} \textsf{c-execute}_n(\lfloor c \rfloor)$$

This follows from (`forall_iff_mul`)

$$\forall\, n \in \mathbb{N}. \qquad \textsf{sc-execute}(c) \gg \kappa \;\dot{\Rrightarrow}\; \kappa \gg \textsf{c-execute}_n(\lfloor c \rfloor)$$

We prove this by induction on n and considering every case for c. Most cases are trivial due to the fact that both executions share the same definitions. Only routine call, open and close require our attention.

The case for routine invocation follows from Lemma 9.9.8. Execution of **open** and **close** must be shown to have no discernable effect if followed by κ. For **open**, we must show that

$$\begin{pmatrix} \textbf{do} & \ell \leftarrow \textsf{evaluate}(e) \\ & p(\ell, v) \leftarrow \textsf{find-chunk}(p, \ell) \\ & \textsf{consume-chunk}(p(\ell, v)) \\ & \textsf{with-store}(s_0[x := \ell][y := v], \textsf{produce-assertion}(a)) \\ & \kappa \end{pmatrix} \dot{\Rrightarrow} \kappa$$

where **predicate** $p(x, y) = a$. Using Lemma 9.9.9, we can rewrite this as

$$\begin{pmatrix} \textbf{do} & \ell \leftarrow \textsf{evaluate}(e) \\ & p(\ell, v) \leftarrow \textsf{find-chunk}(p, \ell) \\ & \textsf{consume-chunk}(p(\ell, v)) \\ & \textsf{produce-chunk}(p(\ell, v)) \\ & \kappa \end{pmatrix} \dot{\Rrightarrow} \kappa$$

It is easy to show that

$$\begin{pmatrix} \textbf{do} & \textsf{consume-chunk}(p(\ell, v)) \\ & \textsf{produce-chunk}(p(\ell, v)) \end{pmatrix} \dot{\Rrightarrow} \textsf{nop}$$

The **close** case follows similarly from Lemma 9.9.9 and Lemma 9.9.5. □

9.10 Symbolic Execution

The semiconcrete execution produces execution trees with finite depth but infinite breadth. The symbolic execution eliminates this last hurdle on our way to a computable verification algorithm.

This execution has two Coq implementations: the scripts are named `Symbolic-Execution.v` and `ESymbolicExecution.v`, the difference being that the latter makes use of the effective result algebra, and is thus fully executable.

9.10.1 Formalisation

The infinite breadth of semiconcrete execution trees are the consequence of demonic choices over $\mathbb{N}$, which occur at several places: memory allocation makes two such choices per memory cell, producing predicate assertions assigns demonically picked values to variables, etc.

Such as a semiconcrete state can be viewed as an abstracted set of a possibly infinite set of concrete states (i.e. heap refinement), a symbolic state will stand for a potentially infinite number of semiconcrete states. We achieve this by introducing *symbols* [75]: a single symbol represents a demonic choice over $\mathbb{N}$.

Example 9.10.1. *The following semiconcrete result*

$$\bigotimes_{\ell \in \mathbb{N}} \bigotimes_{v \in \mathbb{N}} \langle s_0[\mathrm{x} := \ell], \{\ell \mapsto v\}\rangle_{\mathrm{sc}}$$

can be represented by a single symbolic state

$$[\![\langle s_0[\mathrm{x} := \hat{a}], \{\hat{a} \mapsto \hat{b}\}\rangle_{\mathrm{s}}]\!]$$

where $\hat{a}$ and $\hat{b}$ are symbols.

Example 9.10.2. *When allocating a new memory cell, the semiconcrete execution demonically picks a value from $\mathbb{N}$ as the initial contents for this cell. As explained before, this consists of forking execution into an infinite number of paths, one for each $n \in \mathbb{N}$. The symbolic execution avoids this by introducing a fresh symbol.*

Let us say we wish to bind the variables x *and* y *to demonically picked values. The semiconcrete execution would model this as*

```
do $v_1 \leftarrow \mathsf{pick}_{\mathrm{d}}(\mathbb{N})$
   $\mathsf{update\text{-}store}(\mathrm{x}, v_1)$
   $v_2 \leftarrow \mathsf{pick}_{\mathrm{d}}(\mathbb{N})$
   $\mathsf{update\text{-}store}(\mathrm{y}, v_2)$
```

Executing this starting in $\langle s_0, \emptyset\rangle_{\text{sc}}$ *leads to*

$$\bigotimes_{v_1 \in \mathbb{N}} \bigotimes_{v_2 \in \mathbb{N}} [\![\langle s_0[\mathrm{x} := v_1][\mathrm{y} := v_2], \emptyset\rangle_{\text{sc}}]\!]$$

The symbolic execution adopts a different approach:

```
do  $v_1 \leftarrow$ fresh-symbol
    update-store($\mathrm{x}, v_2$)
    $v_2 \leftarrow$ fresh-symbol
    update-store($\mathrm{y}, v_2$)
```

It generates a new symbol which embodies the demonic choice. The result thus becomes

$$[\![\langle s_0[\mathrm{x} := \hat{\mathrm{a}}][\mathrm{y} := \hat{\mathrm{b}}], \emptyset\rangle_{\text{s}}]\!]$$

Note that symbols cannot be used as a substitute for angelic choices over $\mathbb{N}$.

While using this approach is sound, it is too coarse to be useful. During semiconcrete execution, entire ranges may be blocked using assumption. We would like to be able to refine the range of values a symbol can represent. For this, we add a third component to the state, namely the path condition, which is a logical formula constraining the symbols' values.

Example 9.10.3. *The following semiconcrete result*

$$\bigotimes_{0<\ell} \bigotimes_{v<256} [\![\langle s_0[\mathrm{p} := \ell], \ell \mapsto v\rangle_{\text{sc}}]\!]$$

represents states where a nonzero variable p *points to a memory cell containing a byte. All these states can be expressed by a single symbolic state:*

$$[\![\langle s_0[\mathrm{p} := \hat{a}], \hat{a} \mapsto \hat{b}, 0 < \hat{a} \wedge \hat{b} < 256\rangle_{\text{s}}]\!]$$

We now proceed to the formalisation of the symbolic execution. As before, we begin by defining the state the symbolic execution will operate upon, along with all concepts it depends on.

Definition 9.10.1 (`Symbol.t`). *The set Symbol is the set of all symbols, of which there are infinitely many. Symbols will be hatted, i.e.* $\hat{a}, \hat{b}, \ldots$, *as will metavariables involving symbols.*

Expressions cannot be fully evaluated anymore as they may contain symbols; for this reason we introduce terms, which can be viewed as unevaluated expressions where variables are replaced with symbols. Definition 9.10.10 shows how to transform an expression into a term.

Definition 9.10.2 (`Term.t`). *We define terms as*

$$\mathit{Term} ::= \mathit{NatLiteral} \mid \mathit{Symbol} \mid \mathit{Term} +_{\mathrm{t}} \mathit{Term} \mid \mathit{Term} -_{\mathrm{t}} \mathit{Term} \mid \mathit{Term} \times_{\mathrm{t}} \mathit{Term}$$

Example 9.10.4. *In semiconcrete execution,*

$$\begin{array}{ll}\mathbf{do} & v \leftarrow \mathsf{pick_d}(\mathbb{N}) \\ & \mathsf{update\text{-}store}(\mathrm{x}, v+1)\end{array}$$

this leads to an infinite number of states, but since the actual value of v is known, we can increment it and bind the result to the variable x. *The equivalent symbolic operator is*

$$\begin{array}{ll}\mathbf{do} & \hat{v} \leftarrow \mathsf{fresh\text{-}symbol} \\ & \mathsf{update\text{-}store}(\mathrm{x}, \hat{v} +_{\mathrm{t}} 1)\end{array}$$

Since $\hat{v}$ holds a symbol, we cannot evaluate $\hat{v}+1$. Instead, we must store this information as a term.

Definition 9.10.3 (`Formula.t`). *We define formulae as*

$$\begin{array}{rcl}\mathit{Formula} & ::= & \mathit{Term} =_{\mathrm{f}} \mathit{Term} \\ & \mid & \mathit{Term} <_{\mathrm{f}} \mathit{Term} \\ & \mid & \mathit{Term} \leq_{\mathrm{f}} \mathit{Term} \\ & \mid & \mathit{Formula} \wedge_{\mathrm{f}} \mathit{Formula} \\ & \mid & \neg_{\mathrm{f}} \mathit{Formula}\end{array}$$

Formulae are related to boolean expressions in the same way terms are related to expressions: they allow us to embed symbols. Definition 9.10.11 describes the translation from boolean expressions to formulae.

Definition 9.10.4 (`SymbolicExecution.SChunk.t`). *A symbolic chunk takes two terms as arguments.*

$$\mathit{SChunk} = \{p(\hat{t}, \hat{t}') \mid p \in \mathit{Predicate}, \hat{t}, \hat{t}' \in \mathit{Term}\}$$

Definition 9.10.5 (path condition). *A* path condition Φ *is represented by a formula.*

$$\mathit{PathCondition} = \mathit{Formula}$$

Definition 9.10.6 (`SymbolicExecution.SStore.t`). *A* symbolic store $\hat{s}$ *maps variable names to terms.*

$$\mathit{SStore} = \mathit{Id} \longrightarrow \mathit{Term}$$

Definition 9.10.7 (`SymbolicExecution.zero_store`). *The* symbolic zero store $\hat{s}_0$ *maps every variable to the term 0.*

$$\hat{s}_0 = \lambda\, x.\, 0$$

Definition 9.10.8 (`SymbolicExecution.SHeap.t`). *A* symbolic heap $\hat{h}$ *is a multiset of symbolic chunks.*

$$\mathit{SHeap} = \mathit{SChunk} \to \mathbb{N}$$

Definition 9.10.9 (`SymbolicExecution.symbolic_state`). *A* symbolic execution state $\hat{\sigma}$ *consists of three components:*

$$\mathit{SState} = \mathit{PathCondition} \times \mathit{SStore} \times \mathit{SHeap}$$

A symbolic state with path condition Φ, symbolic store $\hat{s}$ and symbolic heap $\hat{h}$ is denoted $\langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}}$.

The following two functions translate expressions and boolean expressions into terms and formulae, respectively and correspond to eval-expr (Def. 9.8.7 on page 144) and eval-bexpr (Def. 9.8.8 on page 144).

Definition 9.10.10 (`Term.of_expression`). expr-to-term(s, e) *converts e into a term, looking up variables in the store s.*

$$\begin{array}{rcl}
\text{expr-to-term}(\hat{s}, n) & = & n \\
\text{expr-to-term}(\hat{s}, x) & = & \hat{s}(x) \\
\text{expr-to-term}(\hat{s}, e +_{\mathrm{e}} e') & = & \text{expr-to-term}(\hat{s}, e) +_{\mathrm{t}} \text{expr-to-term}(\hat{s}, e') \\
\text{expr-to-term}(\hat{s}, e -_{\mathrm{e}} e') & = & \text{expr-to-term}(\hat{s}, e) -_{\mathrm{t}} \text{expr-to-term}(\hat{s}, e') \\
\text{expr-to-term}(\hat{s}, e \times_{\mathrm{e}} e') & = & \text{expr-to-term}(\hat{s}, e) \times_{\mathrm{t}} \text{expr-to-term}(\hat{s}, e')
\end{array}$$

Definition 9.10.11 (`Formula.of_boolean_expression`). bexpr-to-term(s, b) *converts b into a formula, looking up variables in the store s.*

$$\begin{array}{rcl}
\text{bexpr-to-formula}(\hat{s}, e =_{\mathrm{b}} e') & = & \text{expr-to-term}(\hat{s}, e) =_{\mathrm{f}} \text{expr-to-term}(\hat{s}, e') \\
\text{bexpr-to-formula}(\hat{s}, e <_{\mathrm{b}} e') & = & \text{expr-to-term}(\hat{s}, e) <_{\mathrm{f}} \text{expr-to-term}(\hat{s}, e') \\
\text{bexpr-to-formula}(\hat{s}, e \leq_{\mathrm{b}} e') & = & \text{expr-to-term}(\hat{s}, e) \leq_{\mathrm{f}} \text{expr-to-term}(\hat{s}, e') \\
\text{bexpr-to-formula}(\hat{s}, b \wedge_{\mathrm{b}} b') & = & \text{bexpr-to-formula}(\hat{s}, b) \wedge_{\mathrm{f}} \\
 & & \text{bexpr-to-formula}(\hat{s}, b') \\
\text{bexpr-to-formula}(\hat{s}, \neg_{\mathrm{b}} b) & = & \neg_{\mathrm{f}}\ \text{bexpr-to-formula}(\hat{s}, b)
\end{array}$$

We now proceed with the definitions of the operators. Although the symbolic execution seems to have only introduced a minor change, most operators need to be redefined. For the sake of clarity, we will reuse the same names. The basic operators introduced in Sect. 9.5 will remain unchanged, as they are execution-agnostic.

Definition 9.10.12 (`SymbolicExecution.s_operator`). *We define the following shorter notation for operators involving symbolic state:*

$$\mathit{SOperator}_A \equiv \mathit{Operator}_A^{\mathit{SState} \to \mathit{SState}}$$

Purely for the sake of completeness as they hide no surprises, we give full definitions for the current-X and set-current-X operators for each of the three symbolic state components.

Definition 9.10.13 (`current store`, `set_current store`). *The* current-store *and* set-current-store *operators get and set the current store, respectively.*

$$\frac{\textsf{current-store} \;:\; \mathit{SOperator}_{\mathit{SStore}}}{\begin{aligned}\textsf{current-store} \;=\; & \textbf{do}\ \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & \quad \textsf{return}\ \hat{s}\end{aligned}}$$

$$\frac{\textsf{set-current-store} \;:\; \mathit{SStore} \rightarrow \mathit{SOperator}_{\mathrm{unit}}}{\begin{aligned}\textsf{set-current-store}(\hat{s}') \;=\; & \textbf{do}\ \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & \quad \textsf{set-current-state}(\langle \Phi, \hat{s}', \hat{h} \rangle_{\mathrm{s}})\end{aligned}}$$

Definition 9.10.14 (`current heap`, `set_current heap`). *The operators* current-heap *and* set-current-heap *get and set the current heap, respectively.*

$$\frac{\textsf{current-heap} \;:\; \mathit{SOperator}_{\mathit{SHeap}}}{\begin{aligned}\textsf{current-heap} \;=\; & \textbf{do}\ \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & \quad \textsf{return}\ \hat{h}\end{aligned}}$$

$$\frac{\textsf{set-current-heap} \;:\; \mathit{SHeap} \rightarrow \mathit{SOperator}_{\mathrm{unit}}}{\begin{aligned}\textsf{set-current-heap}(\hat{h}') \;=\; & \textbf{do}\ \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & \quad \textsf{set-current-state}(\langle \Phi, \hat{s}, \hat{h}' \rangle_{\mathrm{s}})\end{aligned}}$$

Definition 9.10.15 (`current path_condition`, `set_current path_condition`). *The operators* current-path-condition *and* set-current-path-condition *get and set the current path condition, respectively.*

$$\frac{\textsf{current-path-condition} \;:\; \mathit{SOperator}_{\mathit{Formula}}}{\begin{aligned}\textsf{current-path-condition} \;=\; & \textbf{do}\ \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & \quad \textsf{return}\ \Phi\end{aligned}}$$

$$\frac{\textsf{set-current-path-condition} \;:\; \mathit{Formula} \rightarrow \mathit{SOperator}_{\mathrm{unit}}}{\begin{aligned}\textsf{set-current-path-condition}(\Phi') \;=\; & \textbf{do}\ \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & \quad \textsf{set-current-state}(\langle \Phi', \hat{s}, \hat{h} \rangle_{\mathrm{s}})\end{aligned}}$$

The following operators deal with the store. They are identical to their (semi)concrete counterparts except for the fact they deal with terms instead of natural numbers.

Definition 9.10.16 (`SymbolicExecution.read_store`). *The operator* read-store(x) *looks up the term to which x is bound to by the current store.*

$$\frac{\mathsf{read\text{-}store} \quad : \quad Id \rightarrow SOperator_{Term}}{\begin{aligned}\mathsf{read\text{-}store}(x) \quad = \quad & \mathbf{do}\ \hat{s} \leftarrow \mathsf{current\text{-}store} \\ & \quad \mathsf{return}\ \hat{s}(x)\end{aligned}}$$

Definition 9.10.17 (`SymbolicExecution.update_store`). update-store($x, \hat{t}$) *binds x to $\hat{t}$ in the current store*

$$\frac{\mathsf{update\text{-}store} \quad : \quad Id \rightarrow Term \rightarrow SOperator_{\mathrm{unit}}}{\begin{aligned}\mathsf{update\text{-}store}(x, \hat{t}) \quad = \quad & \mathbf{do}\ \hat{s} \leftarrow \mathsf{current\text{-}store} \\ & \quad \mathsf{set\text{-}current\text{-}store}(\hat{s}[x := \hat{t}])\end{aligned}}$$

Definition 9.10.18 (`SymbolicExecution.evaluate`). evaluate(e) *converts the given expression e to an equivalent term.*

$$\frac{\mathsf{evaluate} \quad : \quad Expr \rightarrow SOperator_{Term}}{\begin{aligned}\mathsf{evaluate}(e) \quad = \quad & \mathbf{do}\ \hat{s} \leftarrow \mathsf{current\text{-}store} \\ & \quad \mathsf{return}\ \text{expr-to-term}(\hat{s}, e)\end{aligned}}$$

Definition 9.10.19 (`SymbolicExecution.to_formula`). *The operator* to-formula *converts a boolean expression to a formula.*

$$\frac{\mathsf{to\text{-}formula} \quad : \quad BExpr \rightarrow SOperator_{Formula}}{\begin{aligned}\mathsf{to\text{-}formula}(b) \quad = \quad & \mathbf{do}\ \hat{s} \leftarrow \mathsf{current\text{-}store} \\ & \quad \mathsf{return}\ \text{bexpr-to-formula}(\hat{s}, b)\end{aligned}}$$

Definition 9.10.20 (`SymbolicExecution.with_store`). *The* with-store(s, op) *operator computes op locally in the given store s.*

$$\frac{\mathsf{with\text{-}store} \quad : \quad SStore \rightarrow SOperator_A \rightarrow SOperator_A}{\begin{aligned}\mathsf{with\text{-}store}(s, op) \quad = \quad & \mathbf{do}\ s' \leftarrow \mathsf{current\text{-}store} \\ & \quad \mathsf{set\text{-}current\text{-}store}(s) \\ & \quad r \leftarrow op \\ & \quad \mathsf{set\text{-}current\text{-}store}(s') \\ & \quad \mathsf{return}\ r\end{aligned}}$$

We turn our attention to operators involving the path condition. Assumption and assertion are simple concepts in the concrete and semiconcrete executions: one only needs to evaluate certain expressions and compare results. In the case of the symbolic execution however, things are more complex as we need to deal with symbols as path conditions.

We expect assumptions and assertions to be able to find out whether a certain condition is satisfied. Since the symbolic execution is meant to be computable, this expectation may be unrealistic. As a precaution, we investigate what kind of concession we may make without compromising soundness.

There are two ways to deviate from the expected behaviour of an assume operator: either it can block too little, or too much. Pruning too little constitutes no problem: more failures will happen, but verification soundness is preserved, only completeness suffers. Conversely, pruning too much must be avoided at all cost. For example, an extreme case would be that an assumption decides to prune all paths, making verification "succeed" immediately.

The opposite rules apply to assertion. We can allow it to be overzealous, failing nondiscriminately, but it must not let pass a state which does not satisfy the given condition.

These concessions make it possible for us to rely on an imperfect algorithm to decide whether or not a condition is satisfied or not. We make use of an external SMT solver: given a set of constraints with unknown variables we ask it to determine whether it can find a model, i.e. if it is possible to assign values to these variables in such a way that all constraints are satisfied. We expect it to return one of three values: **SAT** means a model exists, **UNSAT** that is impossible to find a model, and **UNKNOWN** if the solver has been able to prove neither the possibility nor the impossibility of a model. For our purposes, it is important that, assuming the solver is sound, verification remains sound even if the solver returns **UNKNOWN** every time it is queried.

Assumptions extend the path condition. If the path condition becomes unsatisfiable, we know the symbolic state does not represent any semiconcrete states and we can get rid of it. Thus, assumption will feed the path condition to the SMT solver, and if it returns **UNSAT**, execution blocks.

Asserting a condition P consists of checking that P is implied by the current path condition. We can translate this to a satisfiability problem as follows:

$$\Phi \Rightarrow P \qquad \Longleftrightarrow \qquad \neg\Phi \vee P \qquad \Longleftrightarrow \qquad \neg(\Phi \wedge \neg P)$$

Asserting P is thus equivalent with expecting the SMT solver to return **UNSAT** when given $\Phi \wedge \neg P$. If this is not the case, execution fails.

We model the SMT solver as an operator.

Definition 9.10.21 (`SymbolicExecution.smt`). *An SMT solver is available through the following operator:*

$$\mathsf{smt} : \mathit{Formula} \rightarrow \mathit{SOperator}_{\{\mathbf{SAT},\mathbf{UNSAT},\mathbf{UNKNOWN}\}}$$

Definition 9.10.22 (`SymbolicExecution.assume_formula`). assume-formula(ϕ) *adds the formula ϕ to the current state's path condition. States with unsatisfiable path* *conditions are filtered out.*

$$\begin{array}{rcl} \textsf{assume-formula} & : & \mathit{Formula} \to \mathit{SOperator}_{\mathrm{unit}} \\ \hline \textsf{assume-formula}(\phi) & = & \textbf{do}\ \Phi \leftarrow \textsf{current-path-condition} \\ & & \quad r \leftarrow \textsf{smt}(\Phi \wedge \phi) \\ & & \quad \textsf{assume}\ r \neq \textbf{UNSAT} \\ & & \quad \textsf{set-current-path-condition}(\Phi \wedge \phi) \end{array}$$

Definition 9.10.23 (`SymbolicExecution.assume_bexpr`)**.** assume-bexpr(b) *converts the boolean expression b to a formula, and adds it to the path condition. It blocks executions with provably unsatisfiable path conditions.*

$$\begin{array}{rcl} \textsf{assume-bexpr} & : & \mathit{BExpr} \to \mathit{SOperator}_{\mathrm{unit}} \\ \hline \textsf{assume-bexpr}(b) & = & \textbf{do}\ \phi \leftarrow \textsf{to-formula}(b) \\ & & \quad \textsf{assume-formula}(\phi) \end{array}$$

Definition 9.10.24 (`SymbolicExecution.assert_formula`)**.** assert-formula(ϕ) *fails if the current path condition does not imply the given formula ϕ.*

$$\begin{array}{rcl} \textsf{assert-formula} & : & \mathit{Formula} \to \mathit{SOperator}_{\mathrm{unit}} \\ \hline \textsf{assert-formula}(b) & = & \textbf{do}\ \Phi \leftarrow \textsf{current-path-condition} \\ & & \quad r \leftarrow \textsf{smt}(\Phi \wedge_{\mathrm{f}} \neg_{\mathrm{f}}\phi) \\ & & \quad \textsf{assert}\ r = \textbf{UNSAT} \end{array}$$

Definition 9.10.25 (`SymbolicExecution.assert_bexpr`)**.** *The* assert-bexpr(b) *operator takes a boolean expression b, converts it to a formula, and checks that it follows from the path condition. If not, execution fails.*

$$\begin{array}{rcl} \textsf{assert-bexpr} & : & \mathit{BExpr} \to \mathit{SOperator}_{\mathrm{unit}} \\ \hline \textsf{assert-bexpr}(b) & = & \textbf{do}\ \phi \leftarrow \textsf{to-formula}(b) \\ & & \quad \textsf{assert-formula}(\phi) \end{array}$$

We focus on heap related operators. produce-chunk and consume-chunk remain unchanged with respect to the concrete execution, but we redefine them for the sake of completeness.

Definition 9.10.26 (`SymbolicExecution.produce_chunk`)**.** produce-chunk($\hat{\alpha}$) *adds the chunk $\hat{\alpha}$ to the current heap.*

$$\begin{array}{rcl} \textsf{produce-chunk} & : & \mathit{SChunk} \to \mathit{SOperator}_{\mathrm{unit}} \\ \hline \textsf{produce-chunk}(\hat{\alpha}) & = & \textbf{do}\ \hat{h} \leftarrow \textsf{current-heap} \\ & & \quad \textsf{set-current-heap}(\hat{h} \uplus \{\hat{\alpha}\}) \end{array}$$

Definition 9.10.27 (SymbolicExecution.consume_chunk)**.** *This operator removes the chunk $\hat{\alpha}$ from the heap. If the heap does not contain this chunk, failure ensues.*

$$\begin{array}{rcl}
\text{consume-chunk} & : & \mathit{SChunk} \rightarrow \mathit{SOperator}_{\text{unit}} \\
\hline
\text{consume-chunk}(\hat{\alpha}) & = & \textbf{do}\ \hat{h} \leftarrow \text{current-heap} \\
 & & \quad \text{assert } \hat{\alpha} \in \hat{h} \\
 & & \quad \text{set-current-heap}(\ \hat{h} - \{\hat{\alpha}\}\ )
\end{array}$$

Finding a chunk on the heap is often necessary, which is the responsibility of find-chunk. Since we are using terms, find-chunk cannot just look for a chunk on the heap for which the first argument matches verbatim with the given term. Instead, it must rely on the SMT solver to determine if two terms are equal.

Definition 9.10.28 (SymbolicExecution.find_chunk)**.** *The* find-chunk$(p, \hat{x})$ *operator finds a heap chunk on the current heap, given a predicate p and a first argument $\hat{a}$.*

$$\begin{array}{rcl}
\text{find-chunk} & : & \mathit{Predicate} \rightarrow \mathit{Term} \rightarrow \mathit{SOperator}_{\mathit{SChunk}} \\
\hline
\text{find-chunk}(p, \hat{t}_1) & = & \textbf{do}\ \hat{h} \leftarrow \text{current-heap} \\
 & & \quad p'(\hat{t}'_1, \hat{t}'_2) \leftarrow \text{pick}_{\text{a}}(\hat{h}) \\
 & & \quad \text{assert}(p = p') \\
 & & \quad \text{assert-formula}(\hat{t}_1 =_{\text{f}} \hat{t}'_1) \\
 & & \quad \text{return } p(\hat{t}'_1, \hat{t}'_2)
\end{array}$$

As discussed earlier, the symbolic execution must avoid demonic picks over $\mathbb{N}$. The following operator, fresh-symbol, takes its place.

Definition 9.10.29 (SymbolicExecution.fresh_symbol,.fresh_symbol_n,.fresh_-symbol_1)**.** *The operator* fresh-symbol *X generates a fresh symbol with respect to the (finite) symbol set X. The operator* fresh-symbol *generates a fresh symbol with respect to the current state. The operator* fresh-symbol-n *n X generates n fresh symbols with respect to the (finite) symbol set X. The operator* fresh-symbol-n *n generates n fresh* *symbols with respect to the current state.*

```
fresh-symbol : $\mathcal{P}_{\mathrm{f}}(\mathit{Symbol}) \to \mathit{SOperator}_{\mathit{Symbol}}$
fresh-symbol $X$ = do let $\varsigma = \text{generate-fresh}(X)$
                      assume-formula $(\varsigma =_{\mathrm{f}} \varsigma)$
                      return $\varsigma$

fresh-symbol : $\mathit{SOperator}_{\mathit{Symbol}}$
fresh-symbol = do $[\varsigma] \leftarrow$ fresh-symbol-n $1$
                  return $\varsigma$

fresh-symbol-n : $\mathbb{N} \to \mathcal{P}_{\mathrm{f}}(\mathit{Symbol}) \to \mathit{SOperator}_{\mathit{Symbol}^{[*]}}$
fresh-symbol-n $0$ $X$ = do return []
fresh-symbol-n (S $n$) $X$ = do $\varsigma \leftarrow$ fresh-symbol $X$
                               $\varsigma s \leftarrow$ fresh-symbol-n $n$ $(X \cup \{\varsigma\})$
                               return $(\varsigma :: \varsigma s)$

fresh-symbol-n : $\mathbb{N} \to \mathit{SOperator}_{\mathit{Symbol}^{[*]}}$
fresh-symbol-n $n$ = do $\sigma \leftarrow$ current-state
                        let $\varsigma s = \text{symbols}(\sigma)$ in
                        fresh-symbol-n $n$ $\varsigma s$
```

We left out the definitions for generate-fresh *and* symbols *as they are particularly tedious and unsurprising. Suffice it to say that the former generates a fresh symbol with respect to its argument and the latter collects all symbols from a given state. We refer the interested reader to the Coq script for full definitions (*`Term.symbols`*,* `Formula.symbols`*,* `SymbolicExecution.symbols`*,* `Fresh.fresh`*, ...).*

The reason we add a $\varsigma =_{\mathrm{f}} \varsigma$ formula to the path condition for every fresh symbol *symb* is simple: we use the path condition to keep track of all generated symbols so that we are certain that fresh-symbol does indeed return a fresh symbol. The following example illustrates how things might go wrong if we omit the assume-formula step.

Example 9.10.5. *We show why we need to use the path condition as a means to keep track of all generated symbols. Consider the following operator:*

```
do $\hat{a} \leftarrow$ fresh-symbol
   $\hat{b} \leftarrow$ fresh-symbol
   update-store$(x, \hat{a})$
   update-store$(y, \hat{b})$
```

Clearly, we expect x and y to be bound to different symbols, but in reality, it is possible (even certain if generate-fresh *is deterministic) that they are bound to the same symbol,* *since $\hat{a}$ has never been added to the state. However, this problem can be easily solved by using* fresh-symbol-n *instead.*

A more insidious example goes as follows:

$$\begin{array}{ll}\textbf{do} & \hat{a} \leftarrow \textsf{fresh-symbol} \\ & \textsf{update-store}(x, \hat{a}) \\ & \textsf{with-store}(\hat{s}_0, op)\end{array}$$

Here, the symbol $\hat{a}$ is indeed added to the state, more specifically, to the store. However, the store is temporarily replaced by $\hat{s}_0$ while executing op. If op needs to generate a fresh symbol, it is possible the same symbol $\hat{a}$ will be generated again as nowhere in the current state does $\hat{a}$ appear.

On a high level, reading and writing to cells happen in the same way as with the (semi)concrete execution. The main difference between the execution is abstracted away by find-chunk.

Definition 9.10.30 (`SymbolicExecution.read_cell`)**.** *The operator* read-cell($\hat{\ell}$) *looks for a chunk $\hat{\ell}' \mapsto \hat{v}$ on the current heap such that $\hat{\ell}$ is provably equal to $\hat{\ell}'$ and returns $\hat{v}$. This corresponds to reading the value located at memory address $\hat{\ell}$.*

$$\begin{array}{rcl}\textsf{read-cell} & : & \mathit{Term} \rightarrow \mathit{SOperator}_{\mathit{Term}} \\ \hline \textsf{read-cell}(\hat{\ell}) & = & \textbf{do}\ \ (\hat{\ell}' \mapsto \hat{v}) \leftarrow \textsf{find-chunk}(\mapsto, \hat{\ell}) \\ & & \qquad \textsf{return}\ \hat{v}\end{array}$$

Definition 9.10.31 (`SymbolicExecution.write_cell`)**.** write-cell($\hat{\ell}, \hat{v}$) *looks for a chunk $\hat{\ell}' \mapsto \hat{v}'$ on the current heap such that $\hat{\ell}$ is provably equal to $\hat{\ell}'$. It then removes this chunk and replaces it by $\hat{\ell}' \mapsto \hat{v}$. This corresponds to overwriting the value at memory location $\hat{\ell}$ with $\hat{v}$.*

$$\begin{array}{rcl}\textsf{write-cell} & : & \mathit{Term} \rightarrow \mathit{Term} \rightarrow \mathit{SOperator}_{\text{unit}} \\ \hline \textsf{write-cell}(\hat{\ell}, \hat{v}') & = & \textbf{do}\ \ \hat{\alpha} \leftarrow \textsf{find-chunk}(\mapsto, \hat{\ell}) \\ & & \qquad \textsf{consume-chunk}(\hat{\alpha}) \\ & & \qquad \textsf{produce-chunk}(\hat{\ell} \mapsto \hat{v}')\end{array}$$

The following operators aid in defining memory allocation and deallocation. The following operator, allocate, differs with its (semi)concrete counterpart in that it uses symbols and terms to represent the memory locations where the heap cells are allocated: instead of demonically picking an address ℓ from $\mathbb{N}$ (which leads to infinite branching) and allocating the cells at ℓ, $\ell + 1$, $\ell + 2, \ldots,$ $\ell + n - 1$, the symbolic execution just generates a fresh symbol $\hat{\ell}$ (hence no infinite branching) and places the newly allocated cells at $\hat{\ell} +_t 0, \hat{\ell} +_t 1, \hat{\ell} +_t 2, \ldots, \hat{\ell} +_t (n-1)$.

Take note of the difference between + and $+_t$: 5 + 3 coincides with natural number 8, whereas $5 +_t 3$ is a term, i.e. an unevaluated mathematical expression represented by an abstract syntax tree data structure.

Definition 9.10.32 (`SymbolicExecution.allocate`). *The operator* $\textsf{allocate-at}(\hat{\ell}, n)$ *produces the heap chunks* $(\hat{\ell} +_t i) \mapsto \hat{v}_i$ *for* $i = 0 \ldots n-1$ *and all* v_i *are fresh symbols. The* $\textsf{allocate}(n)$ *picks a random memory location* $\hat{\ell}$ *using a fresh symbol, generates the* $\mapsto$ *chunks by relying on* allocate-at *and returns* $\hat{\ell}$*.*

$$\begin{array}{rcl}
\textsf{allocate-at} & : & \mathit{Symbol} \to \mathit{Symbol}^{[*]} \to \mathit{SOperator}_{\text{unit}} \\
\hline
\textsf{allocate-at}(\hat{\ell}, []) & = & \textbf{do}\ \textsf{nop} \\
\textsf{allocate-at}(\hat{\ell}, \hat{v} :: \hat{vs}) & = & \textbf{do}\ \textsf{produce-chunk}(\hat{\ell} +_t \|\hat{vs}\| \mapsto \hat{v}) \\
 & & \phantom{\textbf{do}\ }\textsf{allocate-at}(\hat{\ell}, \hat{vs})
\end{array}$$

$$\begin{array}{rcl}
\textsf{allocate} & : & \mathbb{N} \to \mathit{SOperator}_{\mathit{Term}} \\
\hline
\textsf{allocate}(n) & = & \textbf{do}\ (\hat{\ell} :: \hat{vs}) \leftarrow \textsf{fresh-symbol-n}(\textsf{n} + 1) \\
 & & \phantom{\textbf{do}\ }\textsf{allocate-at}(\hat{\ell}, \hat{vs}) \\
 & & \phantom{\textbf{do}\ }\textsf{return}\ \hat{\ell}
\end{array}$$

The differences between the (semi)concrete and symbolic versions of the operator consume-cells and *opblocksize* are fully abstracted away by the other lower level operators, i.e. the definitions do not differ from its (semi)concrete variants.

Definition 9.10.33 (`SymbolicExecution.consume_cells`). *The* $\textsf{consume-cells}(\hat{\ell}, n)$ *operator does the opposite of* allocate-at*: it removes one* $\hat{\ell}'_i \mapsto \hat{v}'$ *chunk from the heap for* $i = 0 \ldots n-1$*, where* $\hat{\ell}'_i$ *is provably equal to* $\hat{\ell} +_t i$*.*

$$\begin{array}{rcl}
\textsf{consume-cells} & : & \mathit{Term} \to \mathbb{N} \to \mathit{SOperator}_{\text{unit}} \\
\hline
\textsf{consume-cells}(\hat{\ell}, 0) & = & \textbf{do}\ \textsf{return}\ \square \\
\textsf{consume-cells}(\hat{\ell}, n+1) & = & \textbf{do}\ (\hat{\ell}' \mapsto \hat{v}) \leftarrow \textsf{find-chunk}(\mapsto, \hat{\ell} +_t n) \\
 & & \phantom{\textbf{do}\ }\textsf{consume-chunk}(\hat{\ell}' \mapsto \hat{v}) \\
 & & \phantom{\textbf{do}\ }\textsf{consume-cells}(\hat{\ell}, n)
\end{array}$$

Definition 9.10.34 (`SymbolicExecution.block_size`). *The operator* $\textsf{block-size}(\hat{\ell})$ *returns the size of the memory block allocated at memory location* ℓ *by looking for an* $\textsf{mb}(\ell', n)$ *chunk on the heap, where* ℓ *is provably equal to* ℓ'*.*

$$\begin{array}{rcl}
\textsf{block-size} & : & \mathit{Term} \to \mathit{SOperator}_{\mathbb{N}} \\
\hline
\textsf{block-size}(\hat{\ell}, 0) & = & \textbf{do}\ \textsf{mb}(\hat{\ell}', n) \leftarrow \textsf{find-chunk}(\textsf{mb}, \hat{\ell}) \\
 & & \phantom{\textbf{do}\ }\textsf{return}\ n
\end{array}$$

The symbolic equivalents of consume-assertion and produce-assertion are also needed. They behave exactly the same as their semiconcrete variants.

Definition 9.10.35 (`SymbolicExecution.consume_assertion`)**.** *The auxiliary operator* consume-assertion(a) *checks that the current state satisfies the given assertion a and removes all involved chunks. It operates similarly to it semiconcrete variant from Def. 9.9.11 (page 155).*

$$
\begin{array}{rcl}
\textsf{consume-assertion} & : & \mathit{Assertion} \rightarrow \mathit{SOperator}_{\text{unit}} \\
\hline
\textsf{consume-assertion}(b) & = & \textbf{do}\ \ \textsf{assert-bexpr}(b) \\[1em]
\textsf{consume-assertion}(p(e, ?x)) & = & \textbf{do}\ \ \begin{array}[t]{l} \hat{t}_1 \leftarrow \textsf{evaluate}(e) \\ p(\hat{t}'_1, \hat{t}'_2) \leftarrow \textsf{find-chunk}(p, \hat{t}_1) \\ \textsf{consume-chunk}\ p(\hat{t}'_1, \hat{t}'_2) \\ \textsf{update-store}(x, \hat{t}'_2) \end{array} \\[1em]
\textsf{consume-assertion}(a_1 \star a_2) & = & \textbf{do}\ \ \begin{array}[t]{l} \textsf{consume-assertion}(a_1) \\ \textsf{consume-assertion}(a_2) \end{array}
\end{array}
$$

$$
\begin{array}{l}
\textsf{consume-assertion}(\textbf{if}\ b\ \textbf{then}\ a_1\ \textbf{else}\ a_2) = \\
\quad \textbf{do}\ \ op \leftarrow \textsf{pick}_{\textsf{d}}\left(\left\{ \begin{array}{r} \textsf{assume-bexpr}(b) \gg \textsf{consume-assertion}(a_1) \\ \textsf{assume-bexpr}(\neg_{\text{b}} b) \gg \textsf{consume-assertion}(a_2) \end{array} \right\}\right) \\
\qquad\ \ op
\end{array}
$$

Definition 9.10.36 (`SymbolicExecution.produce_assertion`)**.** *The auxiliary operator* produce-assertion(a) *transforms the state such that it satisfies the given assertion a. It operates similarly to it semiconcrete variant from Def. 9.9.12 (page 156).*

$$
\begin{array}{rcl}
\textsf{produce-assertion} & : & \mathit{Assertion} \rightarrow \mathit{SOperator}_{\text{unit}} \\
\hline
\textsf{produce-assertion}(b) & = & \textbf{do}\ \ \textsf{assume-bexpr}(b) \\[1em]
\textsf{produce-assertion}(p(e, ?x)) & = & \textbf{do}\ \ \begin{array}[t]{l} \hat{t}_1 \leftarrow \textsf{evaluate}(e) \\ \hat{t}_2 \leftarrow \textsf{fresh-symbol} \\ \textsf{produce-chunk}(p(\hat{t}_1, \hat{t}_2)) \\ \textsf{update-store}(x, \hat{t}_2) \end{array} \\[1em]
\textsf{produce-assertion}(a_1 \star a_2) & = & \textbf{do}\ \ \begin{array}[t]{l} \textsf{produce-assertion}(a_1) \\ \textsf{produce-assertion}(a_2) \end{array}
\end{array}
$$

$$
\begin{array}{l}
\textsf{produce-assertion}(\textbf{if}\ b\ \textbf{then}\ a_1\ \textbf{else}\ a_2) = \\
\quad \textbf{do}\ \ op \leftarrow \textsf{pick}_{\textsf{d}}\left(\left\{ \begin{array}{rcl} \textsf{assume-bexpr}(b) & \gg & \textsf{produce-assertion}(a_1) \\ \textsf{assume-bexpr}(\neg_{\text{b}} b) & \gg & \textsf{produce-assertion}(a_2) \end{array} \right\}\right) \\
\qquad\ \ op
\end{array}
$$

We can now define the actual symbolic execution. Apart from the fact it manipulates symbols and terms instead of natural numbers, it is equivalent to the semiconcrete execution (Fig. 9.7 on page 158).

Definition 9.10.37 (`SymbolicExecution.symbolic_execution`)**.** s-execute(c) *returns the result of the symbolic execution of the given command c. The full definition is given in Fig. 9.8.*

As with semiconcrete execution, the symbolic execution needs to consider every routine separately. The validity of each routine implies the validity of the entire program.

Definition 9.10.38 (`SymbolicExecution.leak_check`)**.** *The operator* leak-check *fails if the heap is not empty.*

$$\begin{array}{lcl} \text{leak-check} & : & \mathit{SOperator}_{\text{unit}} \\ \hline \text{leak-check} & = & \textbf{do}\ h \leftarrow \text{current-heap} \\ & & \quad\ \text{assert } h = \emptyset \end{array}$$

Definition 9.10.39 (`SymbolicExecution.valid_routine`)**.** *A routine r with the following definition*

$$\textbf{routine}\ r(x)\ \textbf{requires}\ a\ \textbf{ensures}\ a' \ = \ c$$

is valid iff

$$\text{valid-routine}_{\text{s}}(r) \equiv \left(\begin{array}{ll} \textbf{do} & \hat{v} \leftarrow \text{fresh-symbol} \\ & \text{update-store}(x, \hat{v}) \\ & \text{produce-assertion}(a) \\ & \text{with-store}(s_0[x := \hat{v}], \text{sc-execute}(c)) \\ & \text{consume-assertion}(a') \\ & \text{leak-check} \end{array} \right) (\langle \hat{s}_0, \emptyset \rangle_{\text{sc}}) \not\Longleftrightarrow \bot$$

Definition 9.10.40 (`SymbolicExecution.valid_program`)**.** *Using the symbolic execution, a program is valid if every routine it contains is valid:*

$$\text{valid-program}_{\text{s}}(P) = \forall\, r \in \text{routines}(P).\ \text{valid-routine}_{\text{s}}(r)$$

9.10.2 Relation with Semiconcrete Execution

Now that all three executions have been formalised, we can briefly compare them. Figure 9.9 shows a summary of each execution's properties.

Section 9.9.2 discussed how the semiconcrete execution relates to the concrete execution: a single semiconcrete state actually represents a possibly infinite

$$
\begin{array}{rl}
\text{s-exec}(x := e) = & \textbf{do}\ \hat{v} \leftarrow \text{evaluate}(e); \text{update-store}(x, \hat{v}) \\[1ex]
\text{s-exec}(x := [e']) = & \textbf{do}\ \hat{\ell} \leftarrow \text{evaluate}(e'); \hat{v} \leftarrow \text{read-cell}(\hat{\ell}) \\
& \phantom{\textbf{do}}\ \text{update-store}(x, \hat{v}) \\[1ex]
\text{s-exec}([e] := e') = & \textbf{do}\ \hat{\ell} \leftarrow \text{evaluate}(x); \hat{v} \leftarrow \text{evaluate}(x') \\
& \phantom{\textbf{do}}\ \text{write-cell}(\hat{\ell}, \hat{v}) \\[1ex]
\text{s-exec}(c; c') = & \textbf{do}\ \text{s-exec}(c); \text{s-exec}(c') \\[1ex]
\text{s-exec}(\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c') = & \textbf{do}\ op \leftarrow \text{pick}_{\text{d}} \left(\left\{ \begin{array}{l} \mathsf{A}(b) \gg \text{s-exec}(c) \\ \mathsf{A}(\neg_{\text{b}} b) \gg \text{s-exec}(c') \end{array} \right\} \right); op \\[1ex]
\text{s-exec}(r(e)) = & \textbf{do}\ \hat{v} \leftarrow \text{evaluate}(e) \\
& \phantom{\textbf{do}}\ \text{with-store}\,(\hat{s}_0[x := v], \mathsf{C}(a) \gg \mathsf{P}(a')) \\
& \text{with } \textbf{routine}\ r(x)\ \textbf{requires}\ a\ \textbf{ensures}\ a' \\[1ex]
\text{s-exec}(x := \textbf{malloc}(n)) = & \textbf{do}\ \hat{\ell} \leftarrow \text{allocate}(n) \\
& \phantom{\textbf{do}}\ \text{produce-chunk}(\text{mb}(\hat{\ell}, n)) \\
& \phantom{\textbf{do}}\ \text{update-store}(x, \hat{\ell}) \\[1ex]
\text{s-exec}(\textbf{free}(e)) = & \textbf{do}\ \hat{\ell} \leftarrow \text{evaluate}(x); n \leftarrow \text{block-size}(\hat{\ell}) \\
& \phantom{\textbf{do}}\ \text{consume-chunk}(\text{mb}(\hat{\ell}, n)) \\
& \phantom{\textbf{do}}\ \text{consume-cells}(\hat{\ell}, n) \\[1ex]
\text{s-exec}(\textbf{open}\ p(e, \_)) = & \textbf{do}\ \hat{\ell} \leftarrow \text{evaluate}(e) \\
& \phantom{\textbf{do}}\ p(\hat{\ell}', \hat{v}') \leftarrow \text{find-chunk}(\mapsto, \hat{\ell}) \\
& \phantom{\textbf{do}}\ \text{consume-chunk}(p(\hat{\ell}', \hat{v}')) \\
& \phantom{\textbf{do}}\ \text{with-store}(\hat{s}_0[x := \hat{\ell}'][y := \hat{v}'], \mathsf{P}(a)) \\
& \text{where } \textbf{predicate}\ p(x, y) = a \\[1ex]
\text{s-exec}(\textbf{close}\ p(e, e')) = & \textbf{do}\ \hat{t} \leftarrow \text{evaluate}(e); \hat{t}' \leftarrow \text{evaluate}(e') \\
& \phantom{\textbf{do}}\ \text{with-store}(\hat{s}_0[x := \hat{t}][y := \hat{t}'], \mathsf{C}(a)) \\
& \phantom{\textbf{do}}\ \text{produce-chunk}(p(\hat{t}, \hat{t}')) \\
& \text{where } \textbf{predicate}\ p(x, y) = a \\[1ex]
\text{s-exec}(\textbf{skip}) = & \textbf{do}\ \text{return}\ \square
\end{array}
$$

P = produce-assertion $\quad$ C = consume-assertion $\quad$ A = assume-bexpr

Figure 9.8: Symbolic Execution

	concrete execution	**semiconcrete execution**	**symbolic execution**
routine contracts		✓	✓
user defined predicates		✓	✓
assertions		✓	✓
ghost commands		✓	✓
symbols			✓
computable			✓
execution tree count	1	1+	1+
execution tree depth	∞	n	n
execution tree branching factor	∞	∞	n

Figure 9.9: Execution Comparison

number of concrete states. Execution of a semiconcrete state corresponds to performing an equivalent execution all concrete states it represents. Similarly, a symbolic state stands for a possibly infinite amount of semiconcrete states.

The semiconcrete execution's purpose is to detect failure during concrete execution, which is reflected in the soundness theorem (Theorem 9.9.1 on page 172):

$$\textsf{sc-execute}(c) \gg \kappa \mathrel{\dot{\Rrightarrow}} \kappa \gg \textsf{c-execute}(\lfloor c \rfloor)$$

A similar soundness theorem needs to be proven about symbolic execution. To state this theorem, it is necessary to first define a "state-unfolding" operator. Just as the refinement operator κ translates a semiconcrete state to a demonic pick over concrete states, the concretisation operator ρ transforms a symbolic state into a demonic pick over semiconcrete states.

$$\mathit{SState} \xrightarrow{\rho} \mathit{SCState} \xrightarrow{\kappa} \mathit{CState}$$

As explained before, a symbolic state uses symbols, each of which represents a demonic pick over $\mathbb{N}$, which can then be fine-tuned using the path condition. The translation from a symbolic state to one specific semiconcrete state can be represented by an *interpretation function* mapping symbols to $\mathbb{N}$, which expresses which symbol is assigned which value.

Definition 9.10.41 (interpretation). *An* interpretation *I is a total function with type signature $\mathit{Symbol} \to \mathbb{N}$. The set of all interpretations is denoted $\mathcal{I}$.*

An interpretation can be applied to terms, formulae, stores, . . . We define each in turn.

Definition 9.10.42. *The* interpretation of a term *t is defined as*

$$\begin{array}{rcl} [\![n]\!]_I & = & n \\ [\![a]\!]_I & = & I(a) \\ [\![t_1 +_{\mathrm{t}} t_2]\!]_I & = & [\![t_1]\!]_I + [\![t_2]\!]_I \\ [\![t_1 -_{\mathrm{t}} t_2]\!]_I & = & [\![t_1]\!]_I - [\![t_2]\!]_I \\ [\![t_1 \times_{\mathrm{t}} t_2]\!]_I & = & [\![t_1]\!]_I \times [\![t_2]\!]_I \end{array}$$

Definition 9.10.43. *The* interpretation of a formula ϕ *is defined as*

$$\begin{array}{rcl} [\![t_1 =_{\mathrm{f}} t_2]\!]_I & = & [\![t_1]\!]_I = [\![t_2]\!]_I \\ [\![t_1 <_{\mathrm{f}} t_2]\!]_I & = & [\![t_1]\!]_I < [\![t_2]\!]_I \\ [\![t_1 \leq_{\mathrm{f}} t_2]\!]_I & = & [\![t_1]\!]_I \leq [\![t_2]\!]_I \\ [\![\phi_1 \wedge_{\mathrm{f}} \phi_2]\!]_I & = & [\![\phi_1]\!]_I \wedge [\![\phi_2]\!]_I \\ [\![\neg_{\mathrm{f}} \phi]\!]_I & = & \neg [\![\phi]\!]_I \end{array}$$

Definition 9.10.44. *The* interpretation of a store *s is defined as*

$$[\![s]\!]_I = \lambda\ x.\ [\![s(x)]\!]_I$$

Definition 9.10.45. *The* interpretation of a chunk α *is defined as*

$$[\![p(t_1, t_2)]\!]_I = p([\![t_1]\!]_I, [\![t_2]\!]_I)$$

Definition 9.10.46. *The* interpretation of a heap *h is defined as*

$$[\![h]\!]_I = \{\ [\![\alpha]\!]_I \mid \alpha \in h\ \}$$

Definition 9.10.47. *The* interpretation of a symbolic state *is defined as*

$$[\![\langle \Phi, s, h\rangle_{\mathrm{s}}]\!]_I = \langle [\![s]\!]_I, [\![h]\!]_I \rangle_{\mathrm{sc}}$$

Given these definitions, it is now possible to define concretisation operators. The operator $\rho_{\mathrm{I}}(I)$ transforms the current symbolic state into a semiconcrete state using the translation defined by I. Note this operator blocks if the interpretation does not satisfy the path condition. The ρ operator demonically picks an interpretation, uses it to translate the current state and returns it.

Definition 9.10.48 (concretisation operators)**.** *The* concretisation operators ρ_{I} *and* ρ *are defined as*

$$\begin{array}{rcl} \rho_{\mathrm{I}} & : & \mathcal{I} \to \mathit{Operator}_{\Box}^{\mathit{SState}\to\mathit{SCState}} \\ \hline \rho_{\mathrm{I}}(I) & = & \left(\begin{array}{l} \textbf{do}\ \langle \Phi, \hat{s}, \hat{h}\rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ \quad \textsf{assume}\ I \models \Phi \\ \quad \textsf{set-current-state}([\![\langle \Phi, \hat{s}, \hat{h}\rangle_{\mathrm{s}}]\!]_I) \end{array} \right) \end{array}$$

$$\begin{array}{rcl} \rho & : & \mathit{Operator}_{\mathcal{I}}^{\mathit{SState}\to\mathit{SCState}} \\ \hline \rho & = & \left(\begin{array}{l} \textbf{do}\ I \leftarrow \textsf{pick}_{\mathrm{d}}(\mathcal{I}) \\ \quad \rho_{\mathrm{I}}(I) \\ \quad \textsf{return}\ I \end{array} \right) \end{array}$$

In order to prove the symbolic execution's soundness, a few lemmas are needed. The following lemma shows how concretisation gets rid of the need for a fresh symbol and is used by Lemma 9.10.2, which shows how introducing a fresh symbol corresponds to a demonic choice over $\mathbb{N}$.

Lemma 9.10.1. *For any $I \in \mathcal{I}$ and $v \in \mathbb{N}^{[n]}$,*

$$\textbf{do}\ \ \varsigma s \leftarrow \textsf{fresh-symbol-n}(n); \rho_{\mathrm{I}}(I[\varsigma s := vs]) \quad \stackrel{\cdot}{\Longleftrightarrow} \quad \rho_{\mathrm{I}}(I)$$

Proof. Unfolding ρ_{I} gives

$$\begin{array}{lll} \ldots \stackrel{\cdot}{\Longleftrightarrow} & \textbf{do} & \varsigma s \leftarrow \textsf{fresh-symbol-n}(n); \\ & & \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & & \textsf{assume}\ (I[\varsigma s := vs] \models \Phi) \\ & & \textsf{set-current-state}(\langle [\![\hat{s}]\!]_{I[\varsigma s := vs]}, [\![\hat{h}]\!]_{I[\varsigma s := vs]} \rangle_{\mathrm{sc}}) \end{array}$$

Since ςs are fresh with respect to Φ, it follows that

$$\begin{array}{rcl} I[\varsigma s := vs]\ \models\ \Phi & \Longleftrightarrow & I \models \Phi \\ [\![\hat{s}]\!]_{I[\varsigma s := vs]} & = & [\![\hat{s}]\!]_I \\ [\![\hat{h}]\!]_{I[\varsigma s := vs]} & = & [\![\hat{h}]\!]_I \end{array}$$

Applying these simplifications results in

$$\begin{array}{lll} \ldots \stackrel{\cdot}{\Longleftrightarrow} & \textbf{do} & \varsigma s \leftarrow \textsf{fresh-symbol-n}(n); \\ & & \langle \Phi, \hat{s}, \hat{h} \rangle_{\mathrm{s}} \leftarrow \textsf{current-state} \\ & & \textsf{assume}\ (I \models \Phi) \\ & & \textsf{set-current-state}(\langle [\![\hat{s}]\!]_I, [\![\hat{h}]\!]_I \rangle_{\mathrm{sc}}) \end{array}$$

The fresh symbols have become useless. Removing them and folding gives

$$\ldots \stackrel{\cdot}{\Longleftrightarrow} \quad \textbf{do}\ \ \rho_{\mathrm{I}}(I)$$

□

Lemma 9.10.2.

$$\left(\begin{array}{ll} \textbf{do} & \varsigma s \leftarrow \textsf{fresh-symbol-n}(n) \\ & I \leftarrow \rho \\ & \textsf{return}\ [\![\varsigma s]\!]_I \end{array} \right) \stackrel{\cdot}{\Rrightarrow} \left(\begin{array}{ll} \textbf{do} & \rho \\ & \textsf{pick}_{\mathrm{d}}(\mathbb{N}^{[n]}) \end{array} \right)$$

Proof. Starting from the lhs, we make use of transitivity by building a $\stackrel{\cdot}{\Rrightarrow}$ chain ending in the rhs.

$$\textbf{do}\ \ \varsigma s \leftarrow \textsf{fresh-symbol-n}(n); I \leftarrow \rho; \textsf{return}\ [\![\varsigma s]\!]_I$$

Unfolding ρ gives

$$\ldots \dot{\Rrightarrow} \ \mathbf{do}\ \ \varsigma s \leftarrow \mathsf{fresh\text{-}symbol\text{-}n}(n); I \leftarrow \mathsf{pick_d}(\mathcal{I}); \rho_{\mathrm{I}}(I); \mathsf{return}\ [\![\varsigma s]\!]_I$$

For any symbols ςs,

$$\mathsf{pick_d}(\mathcal{I}) \iff \mathbf{do}\ \ I \leftarrow \mathsf{pick_d}(\mathcal{I}); vs \leftarrow \mathsf{pick_d}(\mathbb{N}^{[n]}); \mathsf{return}\ I[\varsigma s := vs]$$

Using this equivalence, we get

$$\begin{aligned}\ldots \dot{\Rrightarrow} \ \mathbf{do}\ \ & \varsigma s \leftarrow \mathsf{fresh\text{-}symbol\text{-}n}(n);\\ & I \leftarrow \mathsf{pick_d}(\mathcal{I});\\ & vs \leftarrow \mathsf{pick_d}(\mathbb{N}^{[n]});\\ & \rho_{\mathrm{I}}(I[\varsigma s := vs]);\\ & \mathsf{return}\ vs\end{aligned}$$

Shifting the two demonic picks up is possible as they don't depend on ςs:

$$\begin{aligned}\ldots \dot{\Rrightarrow} \ \mathbf{do}\ \ & I \leftarrow \mathsf{pick_d}(\mathcal{I});\\ & vs \leftarrow \mathsf{pick_d}(\mathbb{N}^{[n]});\\ & \varsigma s \leftarrow \mathsf{fresh\text{-}symbol\text{-}n}(n);\\ & \rho_{\mathrm{I}}(I[\varsigma s := vs]);\\ & \mathsf{return}\ vs\end{aligned}$$

Applying Lemma 9.10.1 transforms this into

$$\ldots \dot{\Rrightarrow} \ \mathbf{do}\ \ I \leftarrow \mathsf{pick_d}(\mathcal{I}); vs \leftarrow \mathsf{pick_d}(\mathbb{N}^{[n]}); \rho_{\mathrm{I}}(I); \mathsf{return}\ vs$$

Shifting the demonic pick for vs back to the right:

$$\ldots \dot{\Rrightarrow} \ \mathbf{do}\ \ I \leftarrow \mathsf{pick_d}(\mathcal{I}); \rho_{\mathrm{I}}(I); vs \leftarrow \mathsf{pick_d}(\mathbb{N}^{[n]}); \mathsf{return}\ vs$$

Folding and simplifying results in

$$\ldots \dot{\Rrightarrow} \ \mathbf{do}\ \ \rho; \mathsf{pick_d}(\mathbb{N}^{[n]})$$

□

We have been overloading certain operator names, i.e. some operators have concrete, semiconcrete and symbolic variants. To clearly distinguish between these different versions, we will add prefixes (c-, sc- and s-) where necessary.

Lemma 9.10.3. *The symbolic assertion production implies the semiconcrete assertion production under concretisation.*

$$\mathsf{s\text{-}produce\text{-}assertion}(a) \ggg \rho \quad \dot{\Rrightarrow} \quad \rho \ggg \mathsf{sc\text{-}produce\text{-}assertion}(a)$$

Proof. By structural induction on a. □

Lemma 9.10.4. *The symbolic assertion consumption implies the semiconcrete assertion consumption under concretisation.*

$$\mathsf{s\text{-}consume\text{-}assertion}(a) \ggcurly \rho \quad \dot{\Rrightarrow} \quad \rho \ggcurly \mathsf{sc\text{-}consume\text{-}assertion}(a)$$

Proof. By structural induction on a. □

Lemma 9.10.5. *The symbolic and semiconcrete versions of* assume-bexpr *(see Def. 9.10.23 on page 180 and Def. 9.8.16 on page 146, respectively*[10]*) are related as follows:*

$$\left(\begin{array}{ll}\mathbf{do} & \mathsf{s\text{-}assume\text{-}bexpr}(b)\\ & \rho\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & \rho\\ & \mathsf{sc\text{-}assume\text{-}bexpr}(b)\end{array}\right)$$

Proof. Unfolding, applying both sides to the symbolic state $\langle \Phi, \hat{s}, \hat{h}\rangle_\mathrm{s}$ and rearranging yields

$$\begin{array}{l}\mathbf{let}\ r = \mathrm{smt}(\Phi \wedge \phi)\ \mathbf{in}\\ S \models \left(\begin{array}{ll}\mathbf{do} & I \leftarrow \mathsf{pick}_\mathrm{d}(\mathcal{I})\\ & \mathsf{set\text{-}current\text{-}state}(\langle [\![\hat{s}]\!]_I, [\![\hat{h}]\!]_I\rangle_\mathrm{sc})\\ & \mathsf{assume}\ r \neq \mathbf{UNSAT}\\ & \mathsf{assume}\ I \models \Phi \wedge \phi\end{array}\right)(\square)\\ \Downarrow \\ S \models \left(\begin{array}{ll}\mathbf{do} & I \leftarrow \mathsf{pick}_\mathrm{d}(\mathcal{I})\\ & \mathsf{set\text{-}current\text{-}state}(\langle [\![\hat{s}]\!]_I, [\![\hat{h}]\!]_I\rangle_\mathrm{sc})\\ & \mathsf{assume}\ I \models \Phi\\ & \mathsf{assume}(\psi(I) = \mathrm{true})\end{array}\right)(\square)\end{array}$$

where $\phi = \text{bexpr-to-formula}(\hat{s}, b)$ and $\psi(I) = \text{eval-bexpr}([\![\hat{s}]\!]_I, b)$. The only differences between both sides are the assume conditions. We need to show that the lhs does not block if the rhs does not either. In other words, the proof goal becomes

$$\begin{array}{c}\neg\Big(\ \mathrm{smt}(\Phi \wedge \phi) \neq \mathbf{UNSAT} \wedge (I \models \Phi \wedge \phi)\ \Big)\\ \Downarrow\\ \neg(I \models \Phi \wedge \psi(I) = \mathrm{true})\end{array}$$

[10]We remind the reader that the concrete and semiconcrete execution share these definitions.

Rewriting gives

$$\begin{gathered}\text{smt}(\Phi \wedge \phi) = \textbf{UNSAT} \vee I \not\models \Phi \vee I \not\models \phi \\ \Downarrow \\ I \not\models \Phi \vee \psi(I) = \text{false}\end{gathered}$$

Note that $I \not\models \phi \Rightarrow \psi(I) = \text{false}$ (by structural induction on b). We split this into three subproofs:

- First subgoal

  $$\text{smt}(\Phi \wedge \phi) = \textbf{UNSAT} \Rightarrow I \not\models \Phi \vee \psi(I) = \text{false}$$

  Assuming the soundness of the SMT solver, we know

  $$\text{smt}(\Phi \wedge \phi) = \textbf{UNSAT} \Rightarrow \forall\, I.\ (I \not\models \Phi) \vee (I \not\models \phi)$$

  A case split on which disjunct holds leads to two trivial subgoals.

- Second subgoal

  $$I \not\models \Phi \Rightarrow I \not\models \Phi \vee \psi(I) = \text{false}$$

  We pick the left disjunct.

- Third subgoal

  $$I \not\models \phi \Rightarrow I \not\models \Phi \vee \psi(I) = \text{false}$$

  We pick the right conjunct.

□

Lemma 9.10.6. *The symbolic and semiconcrete versions of* assert-bexpr *(see Def. 9.10.25 on page 180 and Def. 9.8.17 on page 146, respectively) are related as follows:*

$$\left(\begin{array}{ll}\textbf{do} & \text{s-assert-bexpr}(b) \\ & \rho\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\textbf{do} & \rho \\ & \text{sc-assert-bexpr}(b)\end{array}\right)$$

Proof. Similar to the proof of Lemma 9.10.5. □

Given these lemmas, we are prepared to show how the symbolic execution is related to the semiconcrete execution.

Theorem 9.10.1. *The symbolic execution is sound with respect to the semiconcrete execution.*

$$\text{s-execute}(c) \gg \rho \ \dot{\Rrightarrow}\ \rho \gg \text{sc-execute}(c)$$

Proof. By structural induction on c. Most proofs consist of shifting the $\rho_{\mathrm{I}}(I)$ operator gradually to the left, translating the symbolic operator into an equivalent semiconcrete one at each jump:

$$\begin{aligned}
&\textsf{s-op}_1 \gg \textsf{s-op}_2 \gg \ldots \gg \textsf{s-op}_{n-1} \gg \textsf{s-op}_n \gg \rho\\
\dot{\Rrightarrow}\ &\textsf{s-op}_1 \gg \textsf{s-op}_2 \gg \ldots \gg \textsf{s-op}_{n-1} \gg \rho \gg \textsf{sc-op}_n\\
&\vdots\\
\dot{\Rrightarrow}\ &\rho \gg \textsf{sc-op}_1 \gg \textsf{sc-op}_2 \gg \ldots \gg \textsf{sc-op}_n
\end{aligned}$$

We demonstrate this scheme by applying it on memory allocation, more specifically the operator allocate, used by $c = \mathbf{malloc}(n)$. allocate has a symbolic variant (Def. 9.10.32, page 184) which we will refer to as s-allocate and a semiconcrete version which is identical to the concrete variant (Def. 9.8.20, page 147), which we rename to sc-allocate.

We first show that, for any $\hat{\ell} \in \mathit{Symbol}$, $\hat{vs} \in \mathit{Symbol}^{[*]}$ and $I \in \mathcal{I}$,

$$\textsf{s-allocate-at}(\hat{\ell}, \hat{vs}) \gg \rho_{\mathrm{I}}(I) \dot{\Rrightarrow} \rho_{\mathrm{I}}(I) \gg \textsf{sc-allocate-at}([\![\hat{\ell}]\!]_I, [\![\hat{vs}]\!]_I) \qquad (9.6)$$

We prove this by structural induction on $\hat{vs}$. The base case $\hat{vs} = []$ is trivial: nop $\dot{\Rrightarrow}$ nop. The proof state for the induction step is

$$\frac{\mathit{IH}:\quad \textsf{s-allocate-at}(\hat{\ell}, \hat{vs}) \gg \rho_{\mathrm{I}}(I) \dot{\Rrightarrow} \rho_{\mathrm{I}}(I) \gg \textsf{sc-allocate-at}([\![\hat{\ell}]\!]_I, [\![\hat{vs}]\!]_I)}{\textsf{s-allocate-at}(\hat{\ell}, \hat{v} :: \hat{vs}) \gg \rho_{\mathrm{I}}(I) \dot{\Rrightarrow} \rho_{\mathrm{I}}(I) \gg \textsf{sc-allocate-at}([\![\hat{\ell}]\!]_I, [\![\hat{v} :: \hat{vs}]\!]_I)}$$

Unfolding s-allocate-at and sc-allocate-at in the goal results in the new goal

$$\begin{pmatrix} \mathbf{do} & \textsf{s-produce-chunk}(\hat{\ell} +_{\mathrm{t}} \|\hat{vs}\| \mapsto \hat{v})\\ & \textsf{s-allocate-at}(\hat{\ell}, \hat{vs})\\ & \rho_{\mathrm{I}}(I) \end{pmatrix}$$

$$\dot{\Rrightarrow} \begin{pmatrix} \mathbf{do} & \rho_{\mathrm{I}}(I)\\ & A \leftarrow \textsf{sc-alloc-set}\\ & \textsf{assume}(([\![\hat{\ell}]\!]_I + \|[\![\hat{vs}]\!]_I\|) \notin A)\\ & \textsf{sc-produce-chunk}(([\![\hat{\ell}]\!]_I + \|[\![\hat{vs}]\!]_I\|) \mapsto \hat{v})\\ & \textsf{sc-allocate-at}(\hat{\ell}, \hat{vs}) \end{pmatrix}$$

We can safely drop the second and third line in the **do**-block of the rhs: leaving them out makes the results stricter (i.e. $op \dot{\Rrightarrow} \textsf{assume}(b) \gg op$). Applying the induction hypothesis *IH* on the lhs results in

$$\left(\begin{array}{ll}\mathbf{do} & \textsf{s-produce-chunk}(\hat{\ell} +_{\mathrm{t}} \|\hat{vs}\| \mapsto \hat{v}) \\ & \rho_{\mathrm{I}}(I) \\ & \textsf{sc-allocate-at}([\![\hat{\ell}]\!]_I, [\![\hat{vs}]\!]_I)\end{array}\right)$$

$$\dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & \rho_{\mathrm{I}}(I) \\ & \textsf{sc-produce-chunk}(([\![\hat{\ell}]\!]_I + \|[\![\hat{vs}]\!]_I\|) \mapsto \hat{v}) \\ & \textsf{sc-allocate-at}(\hat{\ell}, \hat{vs})\end{array}\right)$$

Shifting the $\rho_{\mathrm{I}}(I)$ another step up finishes the proof. Proving this last step is trivial.

We take a step back and look at the allocate operators. Our proof goal is

$$\left(\begin{array}{ll}\mathbf{do} & \hat{\ell} \leftarrow \textsf{s-allocate}(n) \\ & I \leftarrow \rho \\ & \textsf{return } [\![\hat{\ell}]\!]_I\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & \rho \\ & \ell \leftarrow \textsf{sc-allocate}(n) \\ & \textsf{return } \ell\end{array}\right)$$

Unfolding yields

$$\left(\begin{array}{ll}\mathbf{do} & (\hat{\ell} :: \hat{vs}) \leftarrow \textsf{s-fresh-symbol-n(n + 1)} \\ & \textsf{s-allocate-at}(\hat{\ell}, \hat{vs}) \\ & I \leftarrow \rho \\ & \textsf{return } [\![\hat{\ell}]\!]_I\end{array}\right) \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & \rho \\ & (\ell :: vs) \leftarrow \textsf{pick}_{\mathrm{d}}(\mathbb{N}^{[n+1]}) \\ & \textsf{sc-allocate-at}(\ell, vs) \\ & \textsf{return } \ell\end{array}\right)$$

We start with the lhs and gradually transform it into the rhs creating a $\dot{\Rrightarrow}$ chain. We unfold ρ:

$$\ldots \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & (\hat{\ell} :: \hat{vs}) \leftarrow \textsf{s-fresh-symbol-n}(n + 1) \\ & \textsf{s-allocate-at}(\hat{\ell}, \hat{vs}) \\ & I \leftarrow \textsf{pick}_{\mathrm{d}}(\mathcal{I}) \\ & \rho_{\mathrm{I}}(I) \\ & \textsf{return } [\![\hat{\ell}]\!]_I\end{array}\right)$$

$\textsf{pick}_{\mathrm{d}}(\mathcal{I})$ is a state independent demonic operator and s-allocate-at is demonic, meaning we can move the former above the latter (Lemma 9.7.1). We can apply (9.6) to also shift $\rho_{\mathrm{I}}(I)$ up:

$$\ldots \dot{\Rrightarrow} \left(\begin{array}{ll}\mathbf{do} & (\hat{\ell} :: \hat{vs}) \leftarrow \textsf{s-fresh-symbol-n}(n + 1) \\ & I \leftarrow \textsf{pick}_{\mathrm{d}}(\mathcal{I}) \\ & \rho_{\mathrm{I}}(I) \\ & \textsf{sc-allocate-at}([\![\hat{\ell}]\!]_I, [\![\hat{vs}]\!]_I) \\ & \textsf{return } [\![\hat{\ell}]\!]_I\end{array}\right)$$

We fold $\mathsf{pick}_\mathrm{d}(\mathcal{I}) \ggg= \rho_\mathrm{I}$ back into $I \leftarrow \rho$ and rearrange:

$$\ldots \dot{\Rrightarrow} \left(\begin{array}{l} \textbf{do}\ \ (\ell :: vs) \leftarrow \left(\begin{array}{l} \textbf{do}\ \ \varsigma s \leftarrow \textsf{s-fresh-symbol-n}(n+1) \\ \quad I \leftarrow \rho \\ \quad \textsf{return}\ [\![\varsigma s]\!]_I \end{array} \right) \\ \quad \textsf{sc-allocate-at}(\ell, vs) \\ \quad \textsf{return}\ \ell \end{array} \right)$$

Lemma 9.10.2 transforms this into

$$\ldots \dot{\Rrightarrow} \left(\begin{array}{l} \textbf{do}\ \ (\ell :: vs) \leftarrow \left(\begin{array}{l} \textbf{do}\ \ \rho \\ \quad \mathsf{pick}_\mathrm{d}(\mathbb{N}^{[n+1]}) \end{array} \right) \\ \quad \textsf{sc-allocate-at}(\ell, vs) \\ \quad \textsf{return}\ \ell \end{array} \right)$$

which can be rewritten as

$$\ldots \dot{\Rrightarrow} \left(\begin{array}{l} \textbf{do}\ \ \rho \\ \quad (\ell :: vs) \leftarrow \mathsf{pick}_\mathrm{d}(\mathbb{N}^{[n+1]}) \\ \quad \textsf{sc-allocate-at}(\ell, vs) \\ \quad \textsf{return}\ \ell \end{array} \right)$$

which coincides with the rhs of the proof goal. Proofs for the other commands rely on Lemma 9.10.3, Lemma 9.10.4, Lemma 9.10.5 and Lemma 9.10.6.. □

9.10.3 Relation with Concrete Execution

The final step of the soundness proof consists of showing that the symbolic execution is a sound approximation of the concrete execution. A full visual overview is shown in Fig. 9.10.

Theorem 9.10.2. *Given a valid program,*

$$\textsf{s-execute}(c) \gg \rho \gg \kappa \quad \dot{\Rrightarrow} \quad \rho \gg \kappa \gg \textsf{c-execute}(c)$$

Proof.

$$\textsf{s-execute}(c) \gg \rho \gg \kappa$$

$$\dot{\Rrightarrow} (\textsf{s-execute}(c) \gg \rho) \gg \kappa$$

Using Theorem 9.10.1,

$$\dot{\Rrightarrow} (\rho \gg \textsf{sc-execute}(c)) \gg \kappa$$

$$\dot{\Rrightarrow} \rho \gg (\textsf{sc-execute}(c) \gg \kappa)$$

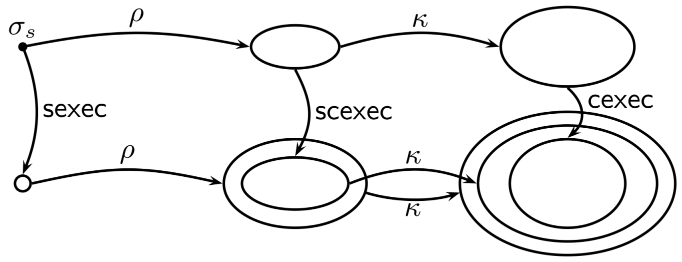

Figure 9.10: Visualisation of the Relation Between the Different Executions

By Lemma 9.9.1,

$$\dot{\Rrightarrow} \rho \gg \kappa \gg \textsf{c-execute}(c)$$

□

Theorem 9.10.3 (main soundness theorem). *Given a valid SIL^{++} program p (see Def. 9.10.40, p. 186) with main routine*

$$\textbf{routine}\ \text{main}(\text{x})\ \textbf{requires}\ \text{true}\ \textbf{ensures}\ \text{true} = c$$

then the concrete execution of $\lfloor p \rfloor$'s main routine will not fail.

$$\forall\, v \in \mathbb{N}. \qquad \textsf{c-execute}(\lfloor c \rfloor)(\langle s_0[\text{x} := v], \emptyset\rangle_{\text{c}}) \not\Leftarrow\!\!\!\Rightarrow \bot$$

Proof. Follows from Theorem 9.10.2. □

9.11 Conclusion

In this chapter, we introduced the reader to Featherweight VeriFast, the theoretical underpinnings of VeriFast. Featherweight VeriFast makes use of an intermediate verification language called Small Imperative Language and is the subject of Sect. 9.1.

A straightforward approach to verifying a SIL program consists of simply executing the program to check that no failures are encountered and is represented by the concrete execution, defined in Sect. 9.8. However, due to the fact that one has to be able to deal with external inputs (demonic choice over an infinite value domain) and nonterminating programs makes this approach computationally infeasible.

To solve this problem, we introduced the symbolic execution (Sect. 9.10) which is a computable approximation to the concrete execution. One of its interesting properties is its reliance on angelic choice to deal with ambiguities during execution, meaning there are now two levels of nondeterminism.

A third execution called the semiconcrete execution (Sect. 9.9) was also defined. It has no other purpose than to serve as a bridge between the concrete and symbolic execution, simplifying the formalisation process.

After having formalised these three executions, we showed how they were related to each other. More specifically, we proved that the symbolic execution can be trusted for the purposes of verification.

These three executions are built on top of a common semantic framework. A first level of abstraction called the result algebra was defined in Sect. 9.3. It offers multiple advantages:

- it allows us to succinctly represent and reason about demonic and angelic choice;
- bottom and top allow us to represent failure and nontermination elegantly: nowhere does the formalisation need to deal with special cases;
- the correspondence between demonic and angelic choice, and universal and existential quantification, respectively, simplifies proofs as it makes it possible to reason abstractly about quantifications;
- it allows us to choose between different models, each having its own pros and cons.

Next, a second, monadic abstraction layer introduces operators which are composable functions returning results (i.e. values from the result algebra). Section 9.5 defined a series of basic operators which serve as building blocks for the formalisation of the semantics of the three executions. They completely abstract away the details of the result algebra.

Many result algebra and operator related lemmas were proven in Coq. The three executions were formalised in Coq, of which the symbolic execution has been defined as an executable function. This means it can be extracted and serve as a core for VeriFast.

Chapter 10

Automation

VeriFast is a powerful static verification tool, which can be used to check a wide range of correctness properties such as memory safety, thread safety, full functional correctness, or even platform specific demands such as JavaCard's transaction system. However, VeriFast requires the programmer to provide annotations, which can amount to a considerable amount of work. Chapter 8 has introduced the reader to most of these annotations. To reiterate:

- Inductive data types allow us to define purely functional data structures. They are used to abstractly represent the contents of C/Java data structures.
- Fixpoint functions are purely functional algorithms operating on inductive data types. Both are often indispensible for proving full functional correctness.
- User defined predicates make it possible to perform heap abstraction, i.e. representing data structures of arbitrary size using a single chunk.
- Lemmas are algorithms necessary to rewrite the symbolic program state.
- The C/Java code often needs to be interspersed with ghost statements, such as **open**, **close** and lemma invocations.

Figure 10.1 contains some statistics. The first column contains the number of lines of C/Java code. The second column shows the annotation line count. Between parenthesis are the number of open and close statements, respectively. The last column gives the number of annotation lines per line of code.

	Lines of Code	Lines of Annotations	Ratio (Ann/Code)
stack (C)	88	198 (18/16)	2.3
sorted binary tree (C)	125	267 (16/23)	2.1
bank example program (C)	405	127 (10/22)	0.31
chat server (C)	130	114 (20/26)	0.88
chat server (Java)	138	144 (19/28)	1.0
game server (Java)	318	225 (47/63)	0.71

Figure 10.1: Annotation Statistics

This chapter discusses ways of reducing the amount of annotations a programmer needs to add to perform verification. Section 10.1 starts with describing our general approach to automation and what benefits follow from it. Next, to be able to demonstrate our automation techniques, we introduce a working example in Sect. 10.2. Three automation techniques are then explained in Sect. 10.3, Sect. 10.4 and Sect. 10.5. We compare them in Sect. 10.6 and conclude in Sect. 10.7.

10.1 General Approach And Rationale

There are many different approaches possible to automation. Some verifiers [12, 45, 28, 116, 27] choose to be fully automated, thus requiring no help from the programmer. However, these verifiers are inherently limited to a small set of verification properties: memory safety is generally within their reach, but achieving full functional correctness is impossible.

An alternative consists of requiring part of the annotations (such as only the function contracts), providing all necessary information about the expected program behaviour, and let the verifier fill in the gaps. Contracts can become very intricate, and it may be a bit naive to expect a verifier to be able to deal with programs of arbitrary complexity. The impossibility of providing the verifier with extra help in cases it fails to verify a piece of code is a major disadvantage.

Our suggested approach consists of strictly separating verification from automation. The verifier should operate on fully annotated code and never make any attempt at inferring annotations. Automation is being taken care of by a separate tool, akin to a preprocessor, which receives code for which it generates as many annotations as possible. The result can then be passed on to the verifier. This technique has several advantages:

- The verifier determines whether a program satisfies certain correctness properties or not. To make it as trustworthy as possible, it should be kept as simple as possible. By strictly limiting it to performing the actual verification, we keep its complexity to a minimum. It thus forms a trusted core, which is small and simple enough to make it feasible to implement it in Coq (or any other proof assistant).
- The separation between verification and automation makes it safe to add new automation techniques without endangering soundness: the generated annotations will always be checked by the verifier in a final step.
- Code given to the automation tool can be partially annotated. This enables programmers to provide extra aid where needed. For example, in case the automation tool has some trouble inferring some loop invariant, the programmer can intervene, specify it himself, and give the code back to the automation tool. Thus, this allows for an iterative (or interactive) approach to verification.

10.2 Working Example

To illustrate the different automation techniques, a working example is necessary. Our object of study will again involve singly linked lists of integer values. Listing 10.1 shows basic definitions:

- A straightforward list struct definition which requires no explanation.
- The inductive type List is used to represent the contents of lists.
- The Node predicate models a single linked list node on the heap. The heap chunk Node(p, q) expresses the fact that a node resides in memory at location p. The node's next pointer equals q but nothing is said about the node's associated integer value.
- An LSeg(p, q) heap chunk reflects the existence of a list segment in memory, i.e. part of a linked list. The first node is pointed to by p, while q points to the node one past the end. For example, LSeg$(p, 0)$ represents a null terminated linked list and LSeg(p, q) and LSeg(q, r) can be merged together to LSeg(p, r).
- The function create_node() creates a new node on the heap.

```
struct list {
  struct list* next;
  int          value;
};

/*@

inductive List = Nil
               | Cons(int, List);

  predicate Node(struct list* P, struct list* Q) =
   P != 0 &*& malloc_block_list(P) &*&
   P->next  |-> Q &*& P->value |-> ?v;

  predicate LSeg(struct list* P, struct list* Q) =
   P == Q ? emp : Node(P, ?R) &*& LSeg(R, Q);

@*/

struct list* create_node();
  //@ requires emp;
  //@ ensures Node(result, 0);
```

Listing 10.1: Basic Definitions for Integer Carrying Singly Linked Lists

The subject of verification is a list copying algorithm, shown in Listing 10.3. As the reader can see, the code requires a relatively large number of allocations: there are approximately twice as many lines of annotations than there are lines of code. Five lemmas were needed to verify[1] the function, whose contracts are shown in Listing 10.2. The meaning of each lemma will be explained while discussing the algorithm.

While understanding the code in Listing 10.3 is no prerequisite for the following sections, we discuss it in detail for the sake of completeness. The function's contract (lines 2–3) only partially describes its copying behaviour: given one linked list, it produces a second list. No guarantees are made that the original list remains unchanged, nor that the new list contains the same elements. While it is possible to prove the function fully functional correct in VeriFast, we have refrained from doing so as it would prevent us to demonstrate the third shape

[1]While this might seem an exorbitant amount of extra work, keep in mind that these lemmas can be reused for verification of functions.

```
lemma void NotNull(struct list* P);
  requires Node(P, ?PN);
  ensures  Node(P, PN) &*& P != 0;

lemma void NoCycle(struct list* P, struct list* Q);
  requires Node(P, Q) &*& LSeg(Q, 0);
  ensures  Node(P, Q) &*& LSeg(Q, 0) &*& P != Q;

lemma void AppendLSeg(struct list* P, struct list* Q);
  requires LSeg(P, Q) &*& Node(Q, ?R) &*&
           Q != R &*& LSeg(R, 0);
  ensures  LSeg(P, R) &*& LSeg(R, 0);

lemma void AppendNode(struct list* P, struct list* Q);
  requires LSeg(P, Q) &*& Node(Q, ?R) &*& Node(R, ?S);
  ensures  LSeg(P, R) &*& Node(R, S);
```

Listing 10.2: Linked List Lemmas

analysis automation technique (Sect. 10.5).

Lines 5–7 deal with empty lists in a straightforward manner. The **close** on line 6 is needed to make the exit state conform with the postcondition. The rest of the function body deals with nonempty lists. The loop forms the heart of the algorithm. The loop invariant can be visualised as follows:

$$\begin{tikzcd}
\mathrm{xs} \arrow[d] & & \mathrm{p} \arrow[d] & \\
\square \arrow[rr, "*"] & & \square \arrow[r, "*"] & \times \\
\square \arrow[r, "*"] & \square \arrow[r] & \times & \\
\mathrm{ys} \arrow[u] & \mathrm{q} \arrow[u] & &
\end{tikzcd}$$

A $\rightarrow^*$ arrow represents an LSeg, i.e. zero or more nodes, while $\rightarrow$ represents a single node. The upper list corresponds to the original list to be copied, while the lower list is the copy under construction. The variables xs and ys at all times point to the first node in the original and copy, respectively. p and q move one step to the right every loop iteration, until p reaches the end of the list. q lags one node behind p.

Lines 8–13 create the copy's first node. Lines 14–19 rewrite the program state so that it matches the loop invariant. The application of the NoCyle lemma

```
struct list* copy(struct list* xs)
 //@ requires LSeg(xs, 0);
 //@ ensures LSeg(xs, 0) &*& LSeg(result , 0);
{
 if ( xs == 0 ) {
  //@ close LSeg(0, 0);                                    // a
  return 0; } else {
  struct list* ys = create_node();
  //@ open LSeg(xs, 0);
  //@ open Node(xs, _);                                    // a
  //@ open Node(ys, 0);                                    // a
  ys->value = xs->value;
  struct list *p = xs->next, *q = ys;
  //@ close Node(ys, 0);                                   // a
  //@ close Node(xs, p);                                   // a
  //@ NoCycle(xs, p);
  //@ close LSeg(p, p);                                    // a
  //@ close LSeg(xs, p);                                   // a
  //@ close LSeg(ys, q);                                   // a
  while ( p != 0 )
    /*@ invariant LSeg(xs,p) &*& LSeg(p,0) &*&
                  LSeg(ys,q) &*& Node(q,0); @*/ {
   //@ struct list *oldp = p, *oldq = q;
   struct list* next = create_node();
   //@ open Node(q, 0);                                    // a
   q->next = next; q = q->next;
   //@ close Node(oldq, q);                                // a
   //@ open LSeg(p, 0);
   //@ assert Node(p, ?pn);
   //@ NoCycle(p, pn);
   //@ open Node(p, _);                                    // a
   //@ open Node(q, 0);                                    // a
   q->value = p->value; p = p->next;
   //@ close Node(q, 0);                                   // a
   //@ close Node(oldp, p);                                // a
   //@ AppendLSeg(xs, oldp); AppendNode(ys, oldq);
  }
  //@ open LSeg(p, 0);                                     // a
  //@ close LSeg(0, 0);                                    // a
  //@ NotNull(q);                                          // b
  //@ AppendLSeg(ys, q);
  //@ open LSeg(0, 0);                                     // a
  return ys;
 }
}
```

Listing 10.3: A VeriFast-Verified List-Copying Algorithm

on line 16 may need more explanation. Applying the lemma informs VeriFast that $\mathrm{xs} \neq \mathrm{p}$. Without it, the **close** on line 18 would not work correctly. When symbolic execution reaches line 18, the heap can be described by

$$\mathrm{Node}(\mathrm{xs}, \mathrm{p}) \star \mathrm{LSeg}(\mathrm{p}, \mathrm{p}) \star \mathrm{LSeg}(\mathrm{p}, 0) \star \mathrm{Node}(\mathrm{ys}, 0)$$

The **close** on line 18 means to fold the first two chunks together in a single chunk $\mathrm{LSeg}(\mathrm{xs}, \mathrm{p})$. However, the LSeg predicate (see Listing 10.1) contains a conditional, leading VeriFast to also consider the possibility where its arguments are equal, i.e. where $\mathrm{xs} = \mathrm{p}$, in which case the heap becomes

$$\mathrm{Node}(\mathrm{xs}, \mathrm{p}) \star \mathrm{LSeg}(\mathrm{p}, \mathrm{p}) \star \mathrm{LSeg}(\mathrm{xs}, \mathrm{p}) \star \mathrm{LSeg}(\mathrm{p}, 0) \star \mathrm{Node}(\mathrm{ys}, 0)$$

In other words, the heap has been extended with the chunk $\mathrm{LSeg}(\mathrm{xs}, \mathrm{p})$, representing an empty list. To prevent VeriFast from considering this path, it is necessary to inform the tool that xs and p cannot be equal. This is achieved by the lemma application on line 16.

Inside the loop, lines 24–35 create a new node and attach it to the end of the copy. When reaching line 36, the heap can be described by

$$\begin{gathered}\mathrm{LSeg}(\mathrm{xs}, \mathrm{oldp}) \star \mathrm{Node}(\mathrm{oldp}, \mathrm{pn}) \star \mathrm{LSeg}(\mathrm{pn}, 0) \star \\ \mathrm{LSeg}(\mathrm{ys}, \mathrm{oldq}) \star \mathrm{Node}(\mathrm{oldq}, \mathrm{q}) \star \mathrm{Node}(\mathrm{q}, 0)\end{gathered}$$

The lemmas on line 36 thus fold the first two chunks on the both lines into a single chunk $\mathrm{LSeg}(\mathrm{xs}, \mathrm{pn})$ and $\mathrm{LSeg}(\mathrm{ys}, \mathrm{q})$, respectively:

$$\mathrm{LSeg}(\mathrm{xs}, \mathrm{pn}) \star \mathrm{LSeg}(\mathrm{pn}, 0) \star \mathrm{LSeg}(\mathrm{ys}, \mathrm{q}) \star \mathrm{Node}(\mathrm{q}, 0)$$

The reader might wonder the reason behind two different lemmas which perform the same task. Looking at the lemma's contract in Listing 10.2, we notice the precondition requires an extra Node or LSeg chunk, depending on the lemma, which seems to remain unused.

Consider the chunks $\mathrm{LSeg}(\mathrm{p}, \mathrm{q})$ and $\mathrm{Node}(\mathrm{q}, \mathrm{r})$. It is not always sound to fold these into $\mathrm{LSeg}(\mathrm{p}, \mathrm{r})$; more specifically, p must not be equal to r. If they were to be equal, we would be dealing with a cyclic list, which cannot be represented by the LSeg predicate. The extra chunk in the precondition provides the lemma with an important piece of information: the separating conjunction guarantees that no node in the list segment pointed to by P is located at memory location R, thus preventing cycles from occurring.

Finally, lines 38–42 take care of transforming the program state left by the loop invariant into one which satisfies the postcondition. The $\mathrm{NotNull}(\mathrm{q})$ lemma application informs VeriFast that q cannot be null, due to the fact that it points to a node in memory. This knowledge is required in order to apply AppendLSeg on line 41: at that point, the heap is

$$\mathrm{LSeg(xs, p)} \star \mathrm{LSeg(ys, q)} \star \mathrm{Node(q, 0)} \star \mathrm{LSeg(0, 0)}$$

If it were possible that q equals 0, AppendLSeg would produce a list with a cyclic node.

10.3 Auto-open and Auto-close

A first automation technique targets the **open** and **close** ghost commands. As can be seen from Fig. 10.1 and Listing 10.3, these comprise a relatively large part of the annotations, which make them ideal candidates for automation.

The idea is simple: whenever symbolic execution fails due to missing heap chunks, attempts are made to find chunks in the current heap which, when opened or folded, produce the missing heap chunks. For example, executing the statement in Listing 10.4 requires a Node(p) chunk on the heap. If it is missing, two possible solutions present themselves:

- The Node(p) chunk might be hidden inside a LSeg(p, q) list segment. Opening up the latter then produces the necessary chunk.
- Conversely, the Node(p)'s component chunks could be present on the heap. Folding them together would also solve the problem.

Instead of randomly opening and closing chunks, the automation tool builds a graph from the predicate definitions which keeps track of the relations between them. Each node corresponds to a predicate and arrows relates a predicate with its constituents. In the case of the predicates defined in Listing 10.1, the graph contains the following nodes and arrows:

$$\begin{array}{ccccc} & \begin{array}{c} a \neq b \\ a = p \end{array} & & p = x & \\ \mathrm{LSeg}(a, b) & \longrightarrow & \mathrm{Node}(p, q) & \longrightarrow & x \to \mathrm{next} \mapsto y \end{array}$$

The arcs are labelled with conditions which the arguments need to satisfy. For example, to rectify a missing Node(p, q) chunk, the heap is searched for an LSeg(a, b) chunk for which the first argument equals the node's ($a = p$). Added to this, the list segment must not be empty ($a \neq b$). If no such LSeg can be found, an $x \to$ next $\mapsto y$ chunk is looked for, and so on.

```
void destroy_node(struct list* p)
  //@ requires Node(p, ?q);
  //@ ensures emp;

destroy_node(p);
```

Listing 10.4: Destroying a Node

For reasons of efficiency, VeriFast restricts its searches to one step deep. For example, a `p->next = 0;` statement, which requires a $x \to \text{next} \mapsto y$ chunk, will succeed if a Node(p, q) node is present, VeriFast will auto-open it. However, an LSeg$(p, 0)$ chunk does not come into consideration.

Using this technique can considerably decrease the amount of necessary annotations. In Listing 10.3, all annotations tagged with "`// a`" are generated automatically: this amounts to 17 out of 31 annotation statements, which is a reduction of than 50%.

10.4 Autolemmas

We now turn our attention to another part of the annotations, namely the lemmas. On the one hand, we have the lemma definitions. For the moment, no efforts have been made to automate this aspect as lemmas need only be defined once, meaning that automatic generation would only yield a limited reduction in annotations.

On the other hand we have the lemma applications, which is where our focus lies. Currently, VeriFast's ability to automatically generate lemma applications is quite specific and admittedly somewhat limited. While automatic opening and closing of predicates is only done when the need arises, VeriFast tries to apply all lemmas regarding a predicate P each time a P chunk is produced, in an attempt to accumulate as much extra information as possible. This immediately gives rise to some obvious limitations:

- It can become quite inefficient: there could be many lemmas to try out and many matches are possible. For example, imagine a lemma operates on a single Node. It can be applied to every Node on the heap, so it is linear with the number of Nodes on the heap. If however it operates on two Nodes, matching becomes quadratic. For this reason, two limitations are imposed: lemmas need to be explicitly declared to qualify for automatic application, and they may only depend on one heap fragment.

- Applying lemmas can modify the execution state so that it becomes unusable. For example, if the AppendLSeg lemma were applied indiscriminately, Nodes would be absorbed by LSegs, effectively throwing away potentially crucial information. In our case, we "forget" that the list segment has length 1. To prevent this, autolemmas are not allowed to modify the symbolic state, but instead may only extend it with extra information.

Given these limitations, in the case of Listing 10.3, only one lemma qualifies for automation: NotNull. Thus, every time a Node(p, q) heap fragment is added to the heap, be it by closing a Node or opening an LSeg or any other way, VeriFast will immediately infer that $p \neq 0$. Since we only needed to apply this lemma once, we decrease the number of annotations by just one line (indicated by "`// b`" in Listing 10.3).

10.5 Shape Analysis

Ideally, we would like the automation tool to generate all annotations. However, it cannot just guess what behaviour a piece of code is meant to exhibit, meaning that it can only check for program independent bugs, such as data races, dangling pointers, etc.

Our third approach for reducing annotations focuses solely on shape analysis [44], i.e. it is limited to checking for memory leaks and invalid pointers dereferences. Fortunately, this limitation is counterbalanced by the fact that it is potentially able to automatically generate all necessary annotations for certain functions. This includes the postcondition, loop invariants, lemma applications, etc.

In order to verify a function by applying shape analysis, we need to determine the initial program state. The simplest way to achieve this is to require the programmer to make his intentions clear by providing preconditions. Even though it appears to be a concession, it has its advantages. Consider the following: the function length requires a list, but last requires a non-empty list. How does the verifier make this distinction? If length contains a bug which makes it fail to verify on empty lists, should the verifier just deduce it is not meant to work on empty lists?

We could have the verifier assume that the buggy length function is in fact correct but not supposed to work on empty lists. The verification is still sound: no memory-related errors will occur. A downside to this approach is that the length function will probably be used elsewhere in the program, and the unnecessary condition of non-emptiness will propagate. At some point, verification will probably fail, but far from the actual location of the bug. Requiring contracts thus puts barriers on how far a bug's influence can reach.

One could make a similar case for the postconditions: shape analysis performs symbolic execution and hence ends up with the final program state. If the programmer provides a postcondition, it can be matched against this final state. This too will prevent a bug's influence from spreading.

Our implementation of shape analysis is based on the approach proposed by Distefano et al. [44]. The idea is simple and very similar to what has been explained earlier in Sect. 10.3: during the symbolic execution of a function, it will open and close the predicates as necessary to satisfy the precondition of the operations it encounters. However, the analysis has a more thorough understanding of the lemmas: it will know in what circumstances they need to be applied. A good example of this is the inference of the loop invariant where shape analysis uses the lemmas to perform state abstraction, which is necessary to prevent the symbolic heap from growing indefinitely while looking for a fixpoint. To clarify, consider the following pseudocode:

$$p' := p; \textbf{while } p \neq 0 \textbf{ do } p := p{\rightarrow}\text{next } \textbf{end}$$

Initially, the symbolic heap contains LSeg(p, 0). To enter the loop, p needs to be non-null, hence it is a non-empty list and can be unfolded to Node(p′, q) ⋆ LSeg(q, 0). During the next iteration, q can be null (the loop ends) or non-null (a second node). Thus, every iteration adds the possibility of an extra node. This way, we'll never find a fixed point.

Performing abstraction will fold nodes back into LSegs, as shown in Fig. 10.2. One might wonder why the abstraction doesn't also merge both LSegs into a single LSeg. The reason for this is that the local variable p points to the start of the second LSeg: folding would throw away information deemed important.

For our purposes, we need to extend the algorithms defined in [44] so that apart from the verification results of a piece of code and final program states which determine the postcondition, they also generate the necessary annotations to be added to the verified code. For example, when performing heap abstraction, i.e. grouping Node and LSeg chunks into a single LSeg, the necessary **close** commands are automatically generated.

Using this approach, the results can be checked by VeriFast, keeping our trusted core to a minimum size (i.e. we do not need to trust the implementation of the

without abstraction	**with abstraction**
$\mathrm{Node}(p', q) \star \mathrm{LSeg}(q, 0)$	$\mathrm{LSeg}(p', q) \star \mathrm{LSeg}(q, 0)$
$\mathrm{Node}(p', q) \star \mathrm{Node}(q, q') \star \mathrm{LSeg}(q', 0))$	$\mathrm{LSeg}(p', q) \star \mathrm{LSeg}(q, 0)$

Figure 10.2: Finding a Fixed Point

shape analysis tool), and extra annotations can be added later on if we wish to prove properties other than memory safety.

In the case of the copy algorithm from Listing 10.3, shape analysis is able to deduce all **open** and **close** annotations, the lemma applications, the loop invariant and the postcondition (in our implementation, we chose to require only the precondition and we manually check that the generated postcondition is as intended). Hence, the number of necessary annotations is reduced to 1, namely the precondition.

10.6 Comparison

In order to get a better idea of by how much we managed to decrease the number of annotations, we wrote a number of list manipulation functions. There are four versions of the code:

(A) A version with all annotations present.

(B) An adaptation of (A) where we enabled auto-open and auto-close (Sect. 10.3).

(C) A version where we take (B) and make `NotNull` an autolemma (Sect. 10.4).

(D) Finally, a minimal version with only the required annotations to make our shape analysis implementation (Sect. 10.5) able to verify the code.

Fig. 10.3 shows how the annotation line counts relate to each other. The left side of the table shows how many annotations were required for the C functions, while the right side considers VeriFast lemmas such as those shown in Listing 10.2. We have omitted their full definitions, but suffice it to say that lemmas also require **open** and **close** statements and can take advantage of autolemma automation as well. There is no (D) column for lemmas as shape analysis has not been adapted to operate on them, although it should not be difficult to achieve this.

Function	**lines of C code**	**(A) no automation**	**(B) auto-open/close**	**(C) auto-open/close/lemma**	**(D) shape analysis**	**Lemma**	**(A) no automation**	**(B) auto-open/close**	**(C) auto-open/close/lemma**
length	10	12	9	9	1	Distinct	9	7	7
sum	11	11	7	7	1	NotNull	7	6	6
destroy	9	6	4	4	1	AppendNode	19	16	16
copy	23	32	15	14	1	AppendLSeg	27	19	18
reverse	12	9	5	5	1	AppendNil	9	7	6
drop_last	28	28	13	13	1	NoCycle	11	10	9
prepend	7	5	3	3	1				
append	13	20	11	11	1				
total	113	205	132	128	8				

Figure 10.3: Annotation Line Count Comparison

The annotation line counts for shape analysis are impressive: the number of required lines of annotations dropped to just 1 (i.e. the precondition). However, this comes at a price: the shape analysis algorithms need to be specialised on a per data structure basis, while auto-open/close and lemmas are universally applicable.

10.7 Conclusion

We can divide verifiers in two categories.

- Fully automatic verifiers which are able to determine whether code satisfies certain conditions without any help of the programmer. Unfortunately, this ease of use comes with a downside: these tools can only check certain properties for certain patterns of code. More ambitious verifications such as ensuring full functional correctness remains out of the scope of these automatic verifiers, since correctness only makes sense with respect to a specification, which needs to be provided by the programmer.

- Non-automatic tools are able to perform more thorough verifications (such as full functional correctness), but these require help from the programmer.

In practice, given a large body of code, it is often sufficient to check only automatically provable properties except for a small section of critical code, where a proof of full functional correctness is necessary. Neither of the above two options is then ideal. Our proposed solution is to combine the best of both worlds by using the following verification framework: at the base lies the non-automatic "core" verifier (in our case VeriFast), which will be responsible for performing the actual verification. To achieve this, it requires code to be fully annotated, but in return, it has the potential of checking for a wide variety of properties. On this base we build an automation layer, consisting of specialised tools able to automatically verify code for specific properties. Instead of just trusting the results of these tools, we require them to produce annotations understood by the core verifier.

A first advantage is that only the core verifier needs to be trusted. Indeed, in the end, all automatically produced annotations are fed back to the core verifier, so that unsoundnesses introduced by buggy automation tools will be caught.

A second advantage is that it allows us to choose which properties are checked for which parts of the code. For example, in order to verify a given program, we would start with unannotated code, on which we would apply an automatic verification tool, such as the shape analysis tool discussed in Sect. 10.5. This produces a number of annotations, which are fed to the core verifier. If verification succeeds, we know the application contains no memory-related errors.

Now consider the case where a certain function foo appears to be troublesome and shape analysis fails to verify it, which could mean that all other parts of the code which call this function also remain unverified. In order to deal with this problem the programmer can manually add the necessary annotations for foo, let the core verifier check them, and then re-apply the shape analysis tool, so that it can proceed with the rest of the code.

After the whole program has been proven memory-safe, one can proceed with the critical parts of the code where a proof of full functional correctness is required. Thus, it makes an iterative incremental approach to verification possible where manually added annotations aid the automatic tools at performing their task.

Chapter 11

Conclusion and Future Work

In this final chapter, we summarise our work (Sect. 11.1), compare the approaches taken by part I and part II (Sect. 11.2), give a short overview of related work (Sect. 11.3) and discuss possible future work (Sect. 11.4).

11.1 Summary

The growing complexity of software (counting millions of lines of code, multithreaded, distributed over multiple machines, fault tolerant, etc.) and our increasing dependence on it (transportation systems, medical devices, power plants, financial institutions, . . .) has made software verification more relevant than ever. In this two part thesis we discussed some existing approaches to software verification, formalised them and proved them sound. A large part of the proofs [110] has also been machine checked with Coq [33, 16].

The first part focused on verification condition generation [42], i.e. given a program's source code, this approach consists of deriving a logic formula from the source code whose validity implies the correctness of the program with respect to its specifications. Three algorithms have been considered: the strongest postcondition (Sect. 5.2), the weakest liberal precondition (Sect. 5.3) and finally the weakest precondition (Sect. 5.4). To prove their soundness, we first formally defined an intermediate verification language (Sect. 4.1) and its semantics (Sect. 4.1.1). Both the semantics and verification condition generation algorithms describe the behaviour of a program. To show the algorithms' soundness, we showed that they are in agreement with the semantics. This has been fully mechanized in Coq: a full implementation and proofs are available at [110].

The weakest preconditions, being the one verifiers (e.g. those built on Boogie [7] or Why3 [49]) generally rely on, enjoyed further development in Chapter 6. The standard algorithm produces verification conditions that grow exponentially with respect to the program size. A more efficient version [52, 77] generates polynomially growing formulae, but only works on passive programs, meaning that the program to be verified needs to be passified prior to the generation of the weakest precondition. We fully formalised this program transformation (Sect. 6.1 and Sect. 6.2) as well as the efficient weakest precondition algorithm (Sect. 6.3). We then proved the soundness of the approach and showed that the generated verification conditions grow polynomially (Sect. 6.4). A full Coq treatment is available at [110].

In part two of this thesis we turned our attention to symbolic execution [75] and separation logic [98]. This approach consists of executing the program and performing the necessary checks at each step, such as ensuring that one has permission to dereference a certain pointer at that moment, etc. Verification succeeds if execution ends without encountering errors. Separation logic was relied upon to deal elegantly with the frame problem.

We defined Featherweight VeriFast (Chapter 9), a minimal verifier named after VeriFast [67], a full-fledged verifier based on symbolic execution and separation logic, of which Featherweight VeriFast is a core part. First, we defined a small intermediate verification language named the “Small Imperative Language” (SIL). To formalise its semantics we used a multilayered semantics framework. The result algebra (Sect. 9.3) constitutes the first abstraction layer, which is used to express the result of a symbolic execution, which involves two kinds of nondeterminism, named angelic and demonic choice. The result algebra allows us to model failure and nontermination elegantly and simplifies the proving process.

As a second abstraction layer we defined operators (Sect. 9.4), i.e. monadic functions returning results (as defined by the result algebra). They are reminiscent of a combination of the well known State and List monads. The binding operation allows us to easily compose operators, automatically taking care of state threading and two kinds of nondeterminism.

Next, we defined a small set of operators, called the basic operators (Sect. 9.5), which form a small domain specific language, abstracting away the details of the result algebra. It is used to formalise the Small Imperative Language’s semantics.

We defined the semantics for SIL in three different ways, named the concrete execution (Sect. 9.8), the semiconcrete execution (Sect. 9.9) and the symbolic execution (Sect. 9.10). The concrete execution is straightforward, but cannot be used for verification as its reliance on infinite sets makes it uncomputable. The symbolic execution is computable thanks to the introduction of symbols, heap abstraction and routine abstraction, but these additions make it more complex. The semiconcrete execution serves as a bridge between the concrete and symbolic execution to simplify the formalisation.

After having defined these three executions, we showed that they were sound approximations of each other for the purposes of verification. In short, if the concrete execution of a program fails, so will the symbolic execution. Thus, the computable symbolic execution can be used to detect failures in the uncomputable concrete execution.

The result algebra, operators, basic operators and three executions were all implemented in Coq [110]. Many low level lemmas were proved in Coq, but due time constraints no fully mechanised version exists of the top level soundness proof. An overview of the sizes of the Coq treatments is shown in Fig. 11.1.

Lastly, we discussed automation for separation logic based verifiers with VeriFast as test subject. VeriFast requires code to be annotated to guide its verification process. We presented and compared three automation techniques. We also proposed a framework which allows us to add new automation methods without putting the soundness of the verification at risk.

11.2 Verification Condition Generation vs Symbolic Execution

In this text, we have presented verification condition generation and symbolic execution as two fundamentally different approaches to software verification, focusing on their dissimilarities. In Sect. 11.2.1, we focus on their similarities. Kassios, Müller and Schwerhoff [73] have also compared how the two approaches perform; we summarise their results in Sect. 11.2.2.

11.2.1 Similarities

Given a procedure to be verified, the symbolic execution approach consists of stepping through it, statement by statement. Each statement makes demands of the current program state, i.e. it has a precondition. If the program state cannot

	Lines of Code (approx)	#Definitions	#Lemmas/Theorems(Fully Proven)
Verification Condition Generation			
Verification Condition Generation (Chapter 5)	1,100	19	36
Efficient Weakest Preconditions (Chapter 6)	7,800	95	202
Symbolic Execution and Separation Logic			
General Purpose	700	25	36
Result Algebra (Sect. 9.3, Sect. 9.4)	3,600	80	130
Result Algebra Models (Sect. 9.6)	900	85	37
Effective Result Algebra	350	50	0
Effective Result Algebra Models	900	80	35
Execution Auxiliary (Sect. 9.1)	1,300	84	12
Executions (Sect. 9.8, Sect. 9.9, Sect. 9.10)	1,400	123	0
Total	18,000	641	488

Figure 11.1: Overview of Coq Formalisation (approximate numbers)

be shown to satisfy this precondition, verification fails. If the precondition is met, the statement transforms the program state in a specific way (i.e. the statement's postcondition). Thus, every statement forms a proof obligation to show that the program state produced by the previous statement satisfies the current statement's precondition.

In other words, each pair of consecutive statements can be seen as generating a verification condition, i.e. an entailment between two separation logic formulae. The verification condition of the entire procedure then becomes the conjunction of all these "local" verification conditions. In practice, symbolic execution cannot proceed after the first failure, forcing us to prove each conjunct in turn instead of all at once. We will come back to this in Sect. 11.2.2, as it has implications for performance.

Stepping back to the level of entire procedures, symbolic execution can be seen as similar to computing the strongest postcondition. Whether it is actually the *strongest* postcondition depends on the symbolic execution's completeness.

A direct correspondence between weakest preconditions and symbolic execution was discussed in Sect. 9.6.2, where we showed that state predicates encoded as sets of sets of states (corresponding to a disjunction of conjunction of states, i.e. a normalised form of the logic formula) forms a model of the result algebra. Operator binding was then shown to be equivalent to the weakest precondition rule for sequencing, and other cases (assume, assert, etc. as in Sect. 4) can be shown to also correspond.

11.2.2 Performance

In a recent paper [73], Kassios et al. compared the performance of verification condition generation and symbolic execution. For their testing, they used the Chalice language [81] in combination with the verification condition generation based verifier Chalice [80] and the symbolic execution based verifier Syxc [102]. Z3 was used as theorem prover. The metrics used were a) time needed to perform verification, b) the number of quantifier instantiations that Z3 had to perform and c) the number of conflicts Z3 encountered during the verification. Their results show that symbolic execution performed on average twice as fast, required 20 times less quantifier instantiations and 5 times less conflicts occurred.

Branching formed a major performance weakness for symbolic execution, a problem we also witnessed with VeriFast. For example, symbolically executing an if-statement leads to a fork in execution, one path for each branch. A series of if-statements then leads to an exponential explosion of branches. This can

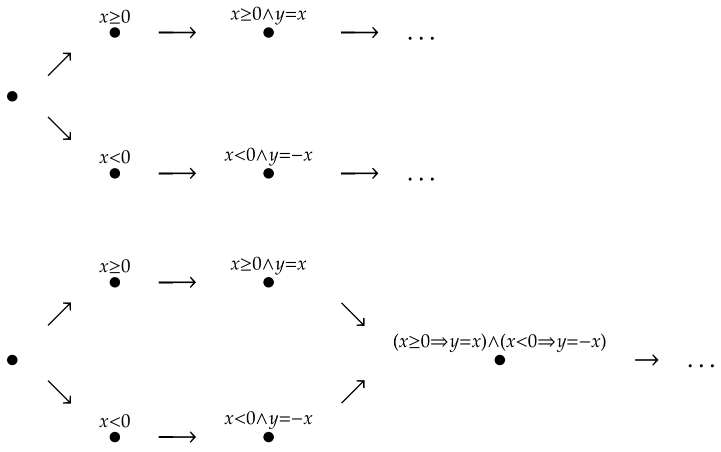

Figure 11.2: Merging Execution Paths

be remedied by merging the two execution paths back together and adding the path-distinguishing information to the path condition, letting the theorem prover (such as Z3) deal with the conditionals. While Kassios et al. found that the theorem prover can deal with conditionals more efficiently, this solution is only applicable whenever the path condition is able to express the difference between the paths.

For example, if the if-branches merely assign different values to variables, such as in

$$\textbf{if } x \geq 0 \textbf{ then } y = x \textbf{ else } y = -x$$

then the two branches can be merged together by adding $(x \geq 0 \Rightarrow y = x) \wedge (x < 0 \Rightarrow y = -x)$ to the path condition (see Fig. 11.2). However, were the if-branches to apply different changes to the heap (e.g. allocate different data structures), then the path condition would not be able to express this.

Another important metric is completeness: are both approaches equally able to recognize correct programs? As explained in the previous section, symbolic execution can be seen as a series of verification conditions, demanding that the current program state implies the precondition of the statement to be executed. Since each conjunct is dealt with separately, not all information is shared between them: only what is expressed by the current symbolic state is known.

For example, consider again singly linked lists (Listing 10.1, page 202). Say the current heap contains the necessary chunks for a list segment, then we can group them into an LSeg using the **close** ghost statement and immediately reopen it with **open**. However, this leads to loss of information as the LSeg predicate forgets about the list segment's size. This in turn may cause verification to fail, even though the code might be correct. Verification condition generation does not suffer from this problem as the verification condition gives the theorem solver a complete view of the situation, not local snapshots.

Related to this issue is the possibility that a symbolic execution based verifier does not feed the theorem prover all information encoded in the symbolic heap. For example, from a heap described by $p \mapsto 0 \star q \mapsto 0$ we can deduce that $p \neq 0$, $q \neq 0$ and $p \neq q$. While in this case this implicit information is trivial, the problem is exacerbated when dealing with fractional permissions [25]. In short, a chunk $p(x, y)$ can be split up into fractions, e.g. $[0.75]p(x, y)$ and $[0.25]p(x, y)$. Adding up all fractions of the same chunk cannot go higher than 1. Thus, a heap described by $[0.60]\mathrm{LSeg}(p, 0) \star [0.30]\mathrm{LSeg}(q, 0) \star [0.80]\mathrm{LSeg}(r, 0)$ implies that $p \neq r$ and $q \neq r$, but p might be equal to q. Thus, a verifier needs to be careful to maximize the amount of information it can deduce from the heap configuration and pass it along to the theorem prover. For example, at the moment of writing, VeriFast[1] fails to take into account fractions.

However, these cases of incompleteness are not inherent to symbolic execution. The LSeg problem discussed first can be solved by defining a new LSeg predicate which keeps track of the list segment size (or even its contents). A more complete set of rules to extract information about the heap will solve the second issue.

Similarly to Chalice and Syxc, VeriCool [104] can be used to compare verification condition generation with symbolic execution as the two approaches are supported by the same tool. The authors also report [103, 105] a performance advantage for symbolic execution. They also found that performance is more predictable (i.e. small changes to specifications generally lead to small changes in performance, contrary to verification condition generation) and simplifies debugging: It can be hard to deduce from information regarding the invalidity of the verification condition where the bug is located, a problem which symbolic execution does not have.

11.3 Related Work

In the following sections, we discuss work related to verification condition generation and symbolic execution.

[1]VeriFast 12.5.23

11.3.1 Verification Condition Generation

Many verifiers rely on verification condition generation. Boogie [7, 18] is a platform upon which several other verifiers have been built. Dafny [78] is an imperative object-based programming language targetting the .NET framework. The language has built-in specification constructs (preconditions, postconditions, loop invariants) which are statically checked using Boogie.

Like Dafny, Chalice [80] defines its own programming language, but specialises in concurrent programs. Chalice programs are verified using verification condition generation, but the same programs can be verified using another verification tool named Syxc [102], which instead uses symbolic execution.

The structure of this thesis might seem to imply that separation logic and verification condition generation are two incompatible approaches. This is certainly not the case, as the VerCors project [2] shows. The VerCors project focuses on the verification of concurrent data structures, checking for data races and functional correctness. For this, it relies on permission based [25, 56] separation logic [98]. For the generation of verification conditions, it uses Chalice [80] and Boogie [7] with plans to also support VeriFast as back end in the future. Currently, two input languages are supported, namely Java and PVL (a small educational language).

HAVOC [76] and VCC [100, 34] check the correctness of annotated C programs. Both have been used in large real world projects: large modules of the Windows operating system have been verified with HAVOC [6] and VCC assisted in the development of Microsoft's hypervisor Hyper-V [31].

Spec$^\sharp$[9] is an extension of the C$^\sharp$ programming language, adding support for non-null types, checked exceptions and method contracts. In a first step, compiling a Spec$^\sharp$ program adds all specifications to the produced assembly in the form of runtime checks. Metadata facilities provided by the .NET CLR [24] are used to tag this extra code so as to make it easy for other tools to distinguish between program and specification. In a second step, a Boogie program is generated from the CIL code [47] from which a verification condition is generated.

Another verification platform is Why [50, 17]. It hosts Jessie [90] (a plugin for the Frama-C environment [53], a framework for static analysis of C programs and is the successor of Caduceus) and Krakatoa, a verifier for Java. Other examples of Java verifiers are JACK [10], ESC/Java [51, 32] and Jahob [23].

Regarding the mechanised verification of verification condition generation algorithms, Homeier and Martin have done similar work to ours. Using the

HOL theorem prover, they also mechanically proved the soundness of the weakest precondition algorithm. In their first work, they focused on partial correctness of an imperative language without support for procedures [61, 62]. In follow up work, they also deal with procedures and mutual recursion [63]. In a third paper [64], they target total correctness, i.e. guaranteed program termination. For this, each procedure is associated with a “variant” expression. They then demand that each cycle in the procedure call graph makes progress, i.e. decreases the value of this variant expression. Note that this allows other procedures in between recursive calls to temporarily undo the progress. Their work resulted in a verified verification condition generator named Sunrise [60].

11.3.2 Symbolic Execution and Separation Logic

Smallfoot [12] is an experimental verifier relying on symbolic execution and separation logic. It operates on a simple imperative programming language and requires annotations in the form of preconditions, postconditions and loop invariants, after which it is able to fully automatically verify the program. This full automation comes at a price: it is restricted to shallow verification properties (i.e. shape analysis, but no full functional correctness) and it only supports a small set of hardcoded predicates (singly-, doubly-, xor-linked lists and trees) meaning it can only deal with a limited set of data structures. Smallfoot can also deal with concurrency, allowing the programmer to call two procedures in parallel and use conditial critical regions [58] as synchronisation primitive.

Smallfoot has been at the center of much further research and has many descendants. A first example is SmallfootRG, which features rely/guarantee reasoning [69], a compositional way of reasoning about concurrent programs. The basic principle is simple: the effects of other threads are abstracted away as possible background interference. $R, G \vdash \{P\}\ c\ \{Q\}$ means that program c transforms a program state satisfying P into one satisfying Q, where R describes all possible interferences from other threads and G the interferences c itself provokes. Two such “R/G-extended Hoare triples” can then be combined into one parallel program if their effects are compatible: $R_1, G_1 \vdash \{P_1\}\ c_1\ \{Q_1\}$ and $R_2, G_2 \vdash \{P_2\}\ c_2\ \{Q_2\}$ lead to $R_1 \cap R_2, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\ c_1 \parallel c_2\ \{Q_1 \wedge Q_2\}$ on condition that $G_1 = R_2$ and $G_2 = R_1$, i.e. if c_2 is impervious to c_1's interferences and vice versa. The condition can of course be weakened to $G_1 \subseteq R_2$ and $G_2 \subseteq R_1$.

Heap-hop [109] is another Smallfoot descendant. It is a verifier for concurrent programs based on Hoare monitors and copyless message passing (i.e. message ownership is lost when sending and this is statically enforced). The tool checks for memory safety, memory leaks, race freedom and deadlock freedom. To achieve this, message channels are given a contract in the form of a finite state machine which describes the protocol to be obeyed.

Smallfoot has (at least) two verified implementations: Tuerk [108] developed Holfoot using the HOL4 [59] theorem prover. It can verify most Smallfoot programs automatically, i.e. it performs shape analysis without programmer help. Contrary to Smallfoot, Holfoot also supports checking for full functional correctness, in which case it produces a series of proof obligations which can then be proved interactively within HOL4 or automatically using an SMT solver.

Appel's VeriSmall [4] is a verified implementation of Smallfoot in Coq [33, 16]. Whereas Holfoot is an *extension* of Smallfoot (as it also allows to check for full functional correctness), VeriSmall is less ambitious and settles for a subset of Smallfoot's capabilities: it can perform shape analysis (like Smallfoot), but limited to singly linked lists.

Unlike Holfoot, which is implemented as a set of inference rules, VeriSmall is written as an algorithm, meaning it can be extracted to a compilable O'Caml program. The verifier finds its application in the Verified Software Toolchain [3], so that it can be used in conjunction with CompCert [83], a certified C compiler.

As mentioned earlier, Smallfoot operates on a small custom programming language, making it unsuitable for use in the real world. Other tools deriving from Smallfoot operate on more pragmatic languages.

SpaceInvader [11, 116], a descendant of Smallfoot, targets C programs. Compared to Smallfoot, it can deal with a wider range of data structures thanks to its use of higher order inductive predicates [11]. In short, the list predicate has been generalised in what type of nodes it has. For example, Listing 8.15 (page 89) shows a List predicate describing a singly linked list, where nodes are described by the Node predicate. Instead of hardcoding the type of the node, we can let List accept a predicate as argument which describes each node. A single List predicate can then be used to represent linked data structures where each node is represented by an arbitrary type of node. The list from Listing 8.15 could then be written List<Node>(p). Further generalisation allows the List predicate to also describe doubly linked lists, cyclic lists or a combination thereof.

Abductor, SpaceInvader's successor, introduced bi-abduction [28]. It allows the tool to also infer preconditions and postconditions. An attempt to explain the basics: since no precondition is known, symbolic execution starts with an empty heap. Execution of a statement makes demands on the current state.

On the one hand, it can lack certain required chunks, and on the other hand, it can contain chunks not needed for the execution of the statement. Written mathematically: $h_{\text{current}} \star \textit{antiframe} \vdash h_{\text{required}} \star \textit{frame}$, where antiframe and frame represent the missing and superfluous chunks, respectively. Biabduction attempts to derive both. This is repeated for every statement in turn. Missing chunks make up the precondition and the final execution state corresponds to the postcondition.

One procedure's contract can depend on those of other procedures. Therefore, it is important to derive the contracts in a stratified manner. To deal with mutually recursive procedures, a fixed point approach is used. Distefano discusses how biabduction performs in the real world in [43]. A more recent biabduction-based tool is Infer [27].

Predator [46] (an open source gcc plugin) and SLAyer [14, 54] are still other verifiers specialised in checking for memory safety and memory leaks in C programs. They are also fully automatic and can deal with complex composite data structures. SLAyer has successfully been applied on real world projects [14] such as device driver verification.

coreStar [20] is a verification framework (similar to Boogie [7] and Why [50, 17]) based on separation logic. It supports two types of symbolic executions: symbolic execution with frame inference (which basically corresponds to standard symbolic execution, i.e. checking that $\{P\}\ c\ \{Q\}$ holds) and a biabductive [28] symbolic execution which, in simplified terms, finds the missing chunks as execution proceeds (i.e. finding a P' such that $\{P \star P'\}\ c\ \{Q\}$ holds). Together these missing chunks form a candidate precondition, which is checked in a second phase by using the frame inferring symbolic execution.

coreStar provides an intermediate verification language named coreStarIL. Compared to other intermediate verification languages, it is quite minimal: a **label** l and matching demonic nondeterministic **goto** $l_1, \ldots, l_n$ command take care of control flow, and a single "specification assignment command" $\overline{x} := \{P\}\{Q\}$ (similar to generic commands [101]) is used for other statements. A fourth and last command, **abs**, triggers abstraction of the current program state, necessary to make symbolic execution converge.

coreStar has no built-in predicates (e.g. there is no $\mapsto$ spatial predicate). Instead, users can provide their own proof rules, rewrite rules and abstraction rules. This is somewhat similar to VeriFast's user defined predicates and (auto)lemmas. Although powerful, a potential problem of this approach is that unsoundnesses can easily be introduced. The authors propose for future versions to either require proofs or to design a specially tailored language which only allows to express rules whose correctness (i.e. confluence and termination) is decidable.

The coreStar platform has been used as backend for multiple verifiers. jStar [45] verifies Java programs (technically, jStar came first and coreStar is a distillation of its core features), MultiStar targets Eiffel and asyncStar [21, 22] specialises in verifying multicore C programs with asynchronous memory operations.

Another example of a symbolic execution and separation logic based verifier is THOR [87], a C verifier which trades in scalability for precision and attempts to combine shape analysis with arithmetic reasoning. In case standard shape analysis fails to prove correctness, a second phase is entered: the heap-manipulating code is translated into arithmetic stack-based code [86] containing "error positions" which during any execution must not be reachable in order for the original code to be correct.

11.4 Future Work

We discuss how the ideas laid out in this thesis can be developed further. Section 11.4.1, Sect. 11.4.2 and Sect. 11.4.3 propose very ad-hoc possibilities for future work, while Sect. 11.4.4 takes a step back and looks at the bigger picture.

11.4.1 Further Formalisation of Verification Condition Generation

Further mechanised formalisation of verification condition generation is possible by also considering unstructured programs and add support for exceptions.

Weakest preconditions for unstructured programs (i.e. programs with gotos) are described in [8]. The verification condition generation happens in several phases, as shown in Fig. 11.3. Intuitively, a program has an irreducible control-flow graph if it contains jumps into loop bodies [1], such as is the case with Duff's device. In a first step, the program must be rewritten into an equivalent form where statements within a loop body can only be reached by paths which pass through the loop's header, i.e. the loop's entry point. In a second step, the back edges (i.e. the jump back from the end of the loop to the loop's entry point) are removed and each loop body is turned in a single "generalized loop iteration" by havocing all loop targets (i.e variables modified in the loop's body) and adding loop invariants. This results in an acyclic control-flow graph, at which point we reach a situation for which the algorithm described in Sect. 6 can take over.

$$\begin{tikzcd}
\text{irreducible control-flow graph} \arrow[d] \\
\text{reducible control-flow graph} \arrow[d] \\
\text{acyclic control-flow graph} \arrow[d] \\
\text{dynamic single-assignment form} \arrow[d] \\
\text{passified form} \arrow[d] \\
\text{weakest precondition}
\end{tikzcd}$$

Figure 11.3: Steps for Weakest Preconditions of Unstructured Programs

Providing support for exceptions [52] is another possible addition. This consists of extending the intermediate verification language with raise and catch statements. The weakest precondition algorithm as discussed in Sect. 5 takes a single target postcondition and returns a weakest precondition. This can be generalized by providing two target postconditions Q and R, which express the conditions that should hold in case of normal and exceptional termination, respectively.

11.4.2 Featherweight VeriFast

The Coq implementation of Featherweight VeriFast is one obvious candidate for further work. As of yet, many proofs about the lower abstraction layers, i.e. the result algebra and operators, have been mechanized (see Fig. 11.1), but much work remains to be done to mechanically prove the soundness of the different executions.

A rather trivial change from a theoretical standpoint, but with large pragmatic consequences, is to extend Featherweight VeriFast with support for predicates and routines with variable number of arguments and return values for the latter. Using SIL^{++} in its current form requires needlessly complicated translations from the source language (e.g. C, Java), whose correctness can then be questioned. A different approach to solve this problem consists of defining yet another intermediate language (SIL$^{\sharp}$) for which we define a translation to SIL^{++}and prove it sound. This both keeps the core language small and simplifies the translation from source language to intermediate verification language.

Some of VeriFast's features such as ghost variables, concurrency and fractional permissions [25] are missing in Featherweight VeriFast. These features add to the verifier's completeness and need to be added to Featherweight VeriFast if it is our intention to be used as VeriFast's certified core without negatively impacting its current verification abilities.

Adding concurrency seems like a particularly interesting problem. The result of the concrete execution would not be a result over states, but over all possible traces. Keeping track of the "execution history" (i.e. a possibly infinite list of program states) should not form a major obstacle: the binding operation should be the only definition that needs modification and the existing operators describing the different execution's semantics should remain unchanged.

However, keeping a single trace does not suffice if we are to model concurrency. The execution of a command c must be seen as the execution in a multithreaded environment where other threads can randomly influence the program state, or at least the heap. Thus, we need to define which execution steps are atomic and consider every possible heap between every such step. A trace would then be a list of functions mapping heaps to states $\tau_i : \mathit{Heap} \to \mathit{Store} \times \mathit{Heap}$. Extracting an actual execution without external interference (required at the top level) then consists of finding a chain where the heaps match at every step: $(s_{i+1}, h_{i+1}) = \tau_i(h_i)$. Most of the changes needed to incorporate this would again take place on the level of operators, with some modifications in the semantics definitions in order to identify atomic steps.

The symbolic execution would need to model each thread separately, the same way every routine is handled in isolation. Rely/Guarantee reasoning in conjunction with separation logic can be used to elegantly restrict the interference from other threads in an abstract manner. We will not delve into this any further as discussing it in detail would probably amount to a whole new chapter.

11.4.3 VeriFast

In its current state, VeriFast does not conform to the framework proposed in Chapter 10: a minimal machine checked core on which other layers are built, but whose results are in a last step always checked by the core, thereby ensuring soundness is preserved.

VeriFast does not make use of a small intermediate verification language, making its core unnecessarily large. The verifier now more or less operates directly on an abstract syntax tree of a hybrid C/Java language. In our opinion, adding an extra indirection in the form of an intermediate verification language constitutes an important step in turning VeriFast into a certified verifier.

As discussed in Chapter 10, VeriFast has some automation techniques built in, such as auto-open/close and autolemmas. However, these are currently implemented within VeriFast's core and leave no trace of their activity. This means a single mistake in their implementation endangers the soundness of the whole verifier.

To improve our confidence in VeriFast's soundness, we would perform the following changes:

- We would refactor VeriFast so that it uses the verifier developed in Chapter 9 as its core.
- The automation techniques mentioned above would be rewritten so as to be restricted to annotation generation instead of directly influencing the symbolic execution. The shape analysis described in Sect. 10.5 already fits this model.
- VeriFast can currently verify C and Java programs. With Featherweight VeriFast as its new core, these would need be translated into SIL^{++} first. The translation algorithm is nontrivial and should itself also be the subject of thorough checking. This can only be done in a meaningful way if there exists a a formal definition of the source language's semantics, so that the original program and its translation can be proven equivalent for the purposes of verification. Ideally, a compiler for the source language should also exist which conforms to this formalised semantics. Examples are the Coq-verified C compiler CompCert [83] and Strecker's [107] work on a verified Java compiler in Isabelle.

Automation can also be improved. For example, the implementation of bi-abduction [28] extended with annotation generation would considerably lighten the programmer's burden as routine contracts and loop invariants could be inferred automatically.

11.4.4 Reflection

We discussed verification condition generation and symbolic execution as general means to perform verification, and Sect. 11.3 presented an (incomplete) list of techniques built on top of these. An obvious question one can ask oneself is, which one is better? It will come as no surprise that no definite answer will be given, for the simple reason that there does not exist a definite answer, and neither should there be one.

To be able to designate which approach to verification is superior to any other, some metric is needed. Many sensible candidates can be found:

- Scalability and efficiency: can the verifier be applied to large programs and give us an answer in reasonable time?
- Difficulty of use: is verification fully automatic, and if not, how much does it impact programmer productivity to verify code? Does the programmer need extra training? Are results easy to understand, i.e. do error messages clearly point out where the bugs hide?
- Correctness guarantees: what type of errors can be caught by the static analysis? Is it limited to type safety or can it prove up to full functional correctness?
- Completeness: dealing with false alarms is time consuming, which negatively affects productivity.
- Applicability: which programming languages does the verifier support? Does it rely on certain programming style to be used?

Unless we manage to develop a verifier scoring high on all these metrics simultaneously, choices need to be made, and just as important, choices need to exist.

How relevant each metric is depends on the situation. If one has an existing codebase written in C, the verifier obviously needs to be able to target C code (directly or indirectly, through translation phases) and probably to detect memory safety and memory leaks at the very least. If one is dealing with a new project which does not need the efficiency of a low level language such as C, it is probably simpler to use a language with garbage collection than to choose one without and rely on a verifier for memory-related correctness properties.

How much verification will break through is an interesting question. Critical pieces of software such as operating systems, drivers and applications involving human life (aircrafts, power plants, . . .) will probably want to make use of the state of art of verification techniques as the costs of failure are great. How much mainstream programs will want to use verification is another matter.

Type systems are already widely in use. Memory safety and memory leaks are no issue in many languages, as most do not allow pointer arithmetic and provide automatic garbage collection. Writing multithreaded code is notoriously hard to do correctly and seems an ideal candidate for verification to prove its usefulness to “everyday programmers”. However, it may be simpler to enforce a certain programming model, such as the actor model, so that fragile parts are taken care of by the language (e.g. Erlang [48], Oz [93, 99]) or by a library (e.g. Scala Actors) and code built on top automatically enjoys certain correctness properties (e.g. data race freedom).

Other possible verification targets are security and privacy related properties. Application distribution systems (e.g. Appstore for iPhone and iPad, Windows Store for Windows 8, app stores for Android) come to mind as they are becoming increasingly popular. Some companies care more than others about the quality and trustworthiness of the available applications and regulate their stores. In the future, they might demand of app developers that their software pass certain tests. At this time, though, it seems that most rules (e.g. regulating access to personal data such as contacts or current location, permission to run in the background, etc.) can be enforced by simple and efficient runtime checks.

The evolution of programming language usage will also influence verifiers. It seems a gradual shift towards a more functional programming style is taking place. Many examples exist: C♯ has recently been extended with many Haskell-like features, such as LINQ and anonymous functions. Java 8 will (finally) also feature lambda functions, as does C++ in its latest incarnation. F♯, closely related to O’Caml, has become a standard part of Visual Studio.

The greater the shift towards the functional paradigm, the more the focus of verification will have to change. For example, functional style often makes use of higher order functions and closures, meaning that objects captured by the closure must be shared or have their ownership rights transferred. Verifiers will need to be able to deal with this elegantly.

Functional style also makes frequent use of immutable objects and data is consequently often shared between different data structures. The immutability could make verification easier, as it makes the framing problem disappear, but is this information used optimally by verifiers [36]? For example, modelling immutable linked lists with node sharing is quite challenging using VeriFast.

With this in mind, one could say we are now contradicting our claims from the introduction where we said that imperative programs are just functional programs with added state, meaning that verification algorithms for imperative languages will just as well be able to operate on functional programs. While this is technically true, such an algorithm does not make use of guarantees implicitly made by functional languages, thus hampering verification somewhat.

Things could also turn out to evolve the other way, away from functional style: research focuses on verification on imperative languages, significant progress is made, the industry starts relying on these tools, with as result that we remain stuck with the same old languages. However, the functional style has gained an advantage the last decade: multithreading has become far more relevant recently, and, in our opinion, the functional style considerably simplifies dealing with multiple threads.

We wondered whether it would be an interesting concept to separate specification from algorithm. Different verifiers use different algorithms, but also require different kinds of annotations. Using a more standard language for specifications could allow multiple verification algorithms to be applied on the same program. For example, Chalice [80] and Syxc [102] use verification condition generation and symbolic execution, respectively, on the same annotated code. Thor [87] views the same program in two different ways: first it looks at it as a heap manipulating program, and in a second phase it abstracts away many details, leaving only a stack based number manipulating program to be verified.

We realised however that this might not turn out to be feasible. The annotation language actually constitutes the main part of a specific technique, as it consists in finding the most common patterns and default values so that, in most cases, specifications remain short and simple while still conveying all necessary information for all different verifications. The different verification algorithms must then make optimal use of this information.

We can more or less define a dependency relation on correctness properties. For example, to prove full functional correctness, many other correctness properties need to be shown to hold, such as the absence of races and deadlocks. This in turns requires us to prove memory safety first. This implies that applying multiple verifiers may not be that useful, as the most advanced verifier will need to prove the same properties as the other less advanced verifiers. This issue can be resolved by letting the verifiers communicate with each other: one verifier could rely on the proof of memory safety, but this information exchange then also needs to be standardised.

Lastly, we discuss the use of proof assistants. During our research, we endeavoured to mechanise as many proofs as possible. One could wonder whether such rigor is actually necessary and if it will occupy a major place in the future.

It is our opinion that the use of proof assistants will certainly grow. Although it is incredibly time consuming, it gives a considerable increase in confidence in the correctness of our definitions and proofs. Some believe a machine checked proof is still not completely trustworthy, as it relies on the soundness of the proof assistant and the correct functioning of the hardware it is running on.

While true, in our eyes, it is still safer than trusting a human expert to check all proofs. Perhaps it is not as trustworthy as a proof checked by a thousand experts, but some pragmatism is required. Also, in our case, not trusting the hardware is not an option as the verified software will end up running on it.

Of course, one has to be careful not to be overconfident: while the proofs are machine checked, the theorems themselves are not. It is possible (and would not be the first time. . .) that one spends hours proving the wrong thing. Hence, human checking is still necessary, but requires far less effort.

Worthy of mention is the POPLmark challenge [5, 97]. It is a set of programming language metatheory problems destined to be mechanised which are meant to stimulate the use of proof assistants. Although it seems interest in it has subsided, we are still confident that mechanisation will keep growing in importance the coming years.

Appendix A

Notations

A.1 Notation definitions

Definition A.1.1 (proof state notation)**.** *During some proofs, we adopt the same proof state notation as Coq.*

$$\begin{array}{l} \textit{Hypothesis 1} \\ \textit{Hypothesis 2} \\ \vdots \\ \textit{Hypothesis n} \\ \hline \textit{Goal} \end{array}$$

Notation	**Short description**	**Definition**	**Page**
$\lambda\ x.\ e$	Lambda function		
$\equiv$	Defined as		
$f[a := b]$	Function update	Def. A.1.2	233
Prop	Type of propositions		
$A^{[*]}$	Type of lists of items A		
$A^{[*]}n$	Type of A-lists of length n		
$\|\|xs\|\|$	Length of list xs		

Figure A.1: General Notations

Notation	Short description
$\wedge$	Conjunction
$\vee$	Disjunction
$\neg$	Negation
$\Rightarrow$	Logically implies
$\Leftarrow$	Logically implied by
$\Longleftrightarrow$	If and only if
$\forall$	Universal quantification
$\exists$	Existential quantification

Figure A.2: Logical Notations

Notation	Short description
$\emptyset$	Empty set
$\mathcal{P}$	Power set
$\mathcal{P}_\mathrm{f}$	Set of all finite subsets
$\in$	Set membership
$\subset$	Strict subset
$\subseteq$	Subset
$\cup$	Union
$\uplus$	Multiset union
$\cap$	Intersection
$-$	Set difference
$\mathbb{N}$	Set of natural numbers, including 0
$\mathbb{N}_0$	Set of natural numbers, excluding 0
$\mathbb{Z}$	Set of integers
unit	Unit set
$\square$	Unit value

Figure A.3: Set Related Notations

Notation	**Short description**	**Definition**	**Page**
Δ_c^P	Class set	Def. 3.2.3	p. 18
Δ_m^P	Method table	Def. 3.2.4	p. 18
Δ_f^P	Field table	Def. 3.2.5	p. 18
Δ^P	Program data	Def. 3.2.8	p. 19
$\rightsquigarrow$	Single step relation arrow	Def. 3.2.11	p. 22
$\rightsquigarrow^*$	Multiple step relation arrow	Def. 3.2.12	p. 25
$\curvearrowright$	Step over relation arrow	Def. 3.2.14	p. 26
$\curvearrowright^*$	Multistep-over relation	Def. 3.2.15	p. 26

Figure A.4: Source Language Notations (Chapter 3)

Notation	**Short description**	**Definition**	**Page**
$[]$	Nondeterministic choice	Def. 4.1.1	p. 31
$\langle c, s \rangle$	Program state	Def. 4.2.4	p. 33
$\longrightarrow$	Single step relation arrow	Def. 4.2.5	p. 35
$\longrightarrow^*$	Multiple step relation arrow	Def. 4.2.7	p. 36
$\downarrow_{\mathrm{f}}$	Big step relation arrow	Def. 5.4.3	p. 50

Figure A.5: Intermediate Verification Language Notations

Notation	**Short description**	**Definition**	**Page**
$\{P\}\ c\ \{Q\}$	Hoare triple	Def. 5.1.2	p. 42
$\lvert P\rvert\ c\ \lvert Q\rvert$	Soft Hoare triple	Def. 5.1.3	p. 42

Figure A.6: Hoare Triples

Notation	**Short description**	**Definition**	**Page**
$[\![e]\!]_v$	Expression versioning	Def. 6.1.2	p. 60
$\sim^v$	Synchronised stores	Def. 6.1.7	p. 61
$\overset{\leq v}{\sim}$	Store equivalence up to v	Def. 6.2.2	p. 63

Figure A.7: Efficient Weakest Precondition Auxiliary Notations

Notation	Short description	Definition	Page
$-^{\star}$	Result	Def. 9.3.1	p. 110
$\oplus$	Angelic choice	Def. 9.3.1	p. 110
$\otimes$	Demonic choice	Def. 9.3.1	p. 110
$\models$	Model	Def. 9.3.1	p. 110
$[\![-]\!]$	Result injection	Def. 9.3.1	p. 110
$\Rrightarrow$	Result implication	Def. 9.3.3	p. 113
$\dot{\Rrightarrow}$	Pointwise result implication	Def. 9.3.5	p. 114
$\Longleftrightarrow$	Result equivalence	Def. 9.3.4	p. 113
$\dot{\Longleftrightarrow}$	Pointwise result equivalence	Def. 9.3.6	p. 114
$\top$	Top	Def. 9.3.7	p. 114
$\bot$	Bottom	Def. 9.3.8	p. 115
$\lvert-\rvert_{\mathrm{d}}$	Characteristic set of demonic result	Def. 9.3.9	p. 118
$\lvert-\rvert_{\mathrm{a}}$	Characteristic set of angelic result	Def. 9.3.10	p. 119
$\lvert-\rvert_{\mathrm{det}}$	Char. value of deterministic result	Def. 9.3.11	p. 119
$-^{\Uparrow}$	Lifting	Def. 9.4.2	p. 121
$\ggg$	Primitive operator binding	Def. 9.4.3	p. 123
$\gg=$	Operator binding	Def. 9.4.6	p. 125
$\gg$	Operator binding (unit result)	Def. 9.4.6	p. 125
do ...	Do notation	Def. 9.4.7	p. 125

Figure A.8: Result Algebra Notations

Notation	Short description	Definition	Page
$+_{\mathrm{e}}\ -_{\mathrm{e}}\ \times_{\mathrm{e}}$	Expression operators	Def. 9.1.1	p. 104
$\mathrm{true}_{\mathrm{b}}\ =_{\mathrm{b}}\ <_{\mathrm{b}}\ \leq_{\mathrm{b}}\ \wedge_{\mathrm{b}}\ \neg_{\mathrm{b}}$	Boolean expr. operators	Def. 9.1.2	p. 104
$\star$	Separating conjunction	Def. 9.9.1	p. 152
$+_{\mathrm{t}}\ -_{\mathrm{t}}\ \times_{\mathrm{t}}$	Term operators	Def. 9.10.2	p. 175
$=_{\mathrm{f}}\ <_{\mathrm{f}}\ \leq_{\mathrm{b}}\ \wedge_{\mathrm{f}}\ \neg_{\mathrm{f}}$	Formula operators	Def. 9.10.3	p. 175
$\mapsto$	Pointer chunk	Def. 9.8.3	p. 143

Figure A.9: SIL/SIL^{++} Notations

Notation	Short description	Definition	Page
$\langle s, h \rangle_{\mathrm{c}}$	Concrete state	Def. 9.8.6	p. 143
$\langle s, h \rangle_{\mathrm{sc}}$	Semiconcrete state	Def. 9.9.9	p. 155
$\langle \hat{s}, \hat{h}, \Phi \rangle_{\mathrm{s}}$	Symbolic state	Def. 9.10.9	p. 176
$- \Vdash -$	Entailment	Def. 9.9.18	p. 161
$[\![\hat{t}]\!]_I$	Interpretation of a term	Def. 9.10.42	p. 189
$[\![\phi]\!]_I$	Interpretation of a formula	Def. 9.10.43	p. 189
$[\![\hat{s}]\!]_I$	Interpretation of a store	Def. 9.10.44	p. 189
$[\![\hat{\alpha}]\!]_I$	Interpretation of a chunk	Def. 9.10.45	p. 189
$[\![\hat{h}]\!]_I$	Interpretation of a heap	Def. 9.10.46	p. 189
$[\![\hat{\sigma}]\!]_I$	Interpretation of a symbolic state	Def. 9.10.47	p. 189
κ	Refinement operator	Def. 9.9.20	p. 162
ρ_{I}	Concretisation operator for I	Def. 9.10.48	p. 189
ρ	Demonic concretisation operator	Def. 9.10.48	p. 189

Figure A.10: Execution Notations

Definition A.1.2 (function update)**.**

$$f[a := b] \equiv \lambda\ x.\ \text{if } x = a \text{ then } b \text{ else } f(x)$$

Appendix B

Monads

This chapter takes on the arduous task of attempting to explain monads to the unsuspecting[1] reader. Monads have been the subject of many tutorials, essays, videos, articles and blogs. This chapter's explanation is based on Brian Beckman's video[2]. Although monads originate from category theory, this introduction does not venture into this highly abstract branch of mathematics but instead focuses on how it is applied in programming languages.

B.1 Basics

If nature were to develop our software, it would randomly generate billions of programs and coldheartedly kill off those that don't perform well. Assuming we had a few eons to spare, we would end up with what would probably be the spaghettiest of code, but it would work. . . most of the time.

We cannot afford to approach software development in the same manner. Instead, we would prefer a more structured method. Abstraction and composability are key ingredients to software engineering: given a limited set of building blocks, we need to be able to compose them, resulting in new larger building blocks, which will themselves serve to build even larger ones.

Composability is paramount. For example, synchronisation (in the context of multithreading) is known to be quite complex. This is partly due to the

[1]The suspecting reader has of course chosen not to read this chapter.
[2]`http://channel9.msdn.com/Shows/Going+Deep/Brian-Beckman-Dont-fear-the-Monads`

fact that locking is still the technique most in use, although it does not offer composability: given two thread safe objects, we cannot just join them together in order to obtain a new thread safe object. It comes at no surprise that transactional memory, an alternative synchronisation method, is receiving much attention recently, as it provides us with a composable way to perform concurrency control.

Functions are probably the most well-known abstraction in programming languages. An important feature is their composability: given two functions f and g, both with type signature int $\rightarrow$ int, we can compose them into a new function $h = g \circ f$, whose signature is again int $\rightarrow$ int. We can generalise this: f and g don't need to specifically operate on integers. In other words, two functions $a \rightarrow a$ can be combined into a new function with the same type signature $a \rightarrow a$, for any type a. Generalizing even further, the input and output types don't need to be the same:

$$f : a \rightarrow b \qquad g : b \rightarrow c \qquad g \circ f : a \rightarrow b$$

We now wish to generalise one step further. We achieve this by introducing a type constructor T and considering functions with type signature $a \rightarrow T\ b$. A type constructor can be seen as a "function on types", comparable to generics (C$^\sharp$, Eiffel, Java, . . .) or templates (C++).

Composing two functions with signatures

$$f : a \rightarrow T\ b \qquad g : b \rightarrow T\ c$$

is not as straightforward anymore, as f's output type does not fit g's input type. For this reason, we need to manually define a *monadic composition*, which we denote $\circ_{\mathrm{m}}$.

$$\circ_{\mathrm{m}} : (b \rightarrow T\ c) \rightarrow (a \rightarrow T\ b) \rightarrow (a \rightarrow T\ c)$$

The monadic composition's responsibility is to take the output of the $a \rightarrow T\ b$ function and somehow apply it on the $b \rightarrow T\ c$ function. The behaviour of monadic composition depends on T: we'll need to define a new monadic composition operation for each different T. In the next section, we'll take a look at different type constructors and how to define their corresponding monadic composition.

B.2 The List Monad

In this section, we take a closer look at the list type constructor. In other words, we wish to compose functions with type signature

$$f : a \rightarrow [b] \qquad g : b \rightarrow [c]$$

where $[b]$ stands for "list of bs". The result of composing f and g should have the type

$$g \circ_{\mathrm{m}} f : a \to [c]$$

The function f returns many bs, while g only accepts a single b. This reminds us of the map function:

$$\mathrm{map} : (a \to b) \to [a] \to [b]$$

The expression map ψ xs applies the function ψ on every element of xs and collects the results in a new list. If we use map directly to compose our functions f and g, i.e.

$$g \circ_{\mathrm{m}} f \overset{?}{=} \mathrm{map}\ g\ \circ f$$

we get the wrong result type: $a \to [[c]]$ instead of $a \to [c]$. We solve this by adding an extra call to concat, which joins all lists together.

$$g \circ_{\mathrm{m}} f = \mathrm{concat}\ \circ\ \mathrm{map}\ g\ \circ f$$

Example B.2.1. *Consider the following functions:*

$$\begin{array}{rcl} \mathrm{double}(n) & = & [n, 2n] \\ \mathrm{clamp}(n) & = & [n \bmod 5] \\ \text{remove-odd}(n) & = & \textbf{if } n \textit{ is odd } \textbf{then } [] \textbf{ else } [n] \end{array}$$

Composing double *with* clamp *amounts to applying* clamp *to every value returned by* double*:*

$$\begin{array}{rcl} (\mathrm{clamp} \circ_{\mathrm{m}} \mathrm{double})(3) & = & [3, 1] \\ (\mathrm{clamp} \circ_{\mathrm{m}} \mathrm{double})(4) & = & [4, 3] \\ (\mathrm{clamp} \circ_{\mathrm{m}} \mathrm{double})(n) & = & [n \bmod 5, 2n \bmod 5] \end{array}$$

We can chain multiple double*s together:*

$$\begin{array}{rcl} (\mathrm{double} \circ_{\mathrm{m}} \mathrm{double})(1) & = & [1, 2, 2, 4] \\ (\mathrm{double} \circ_{\mathrm{m}} \mathrm{double})(n) & = & [n, 2n, 2n, 4n] \\ (\mathrm{double} \circ_{\mathrm{m}} \mathrm{double} \circ_{\mathrm{m}} \mathrm{double})(1) & = & [1, 2, 2, 4, 2, 4, 4, 8] \\ (\mathrm{double} \circ_{\mathrm{m}} \mathrm{double} \circ_{\mathrm{m}} \mathrm{double})(n) & = & [n, 2n, 2n, 4n, 2n, 4n, 4n, 8n] \\ (\mathrm{double} \circ_{\mathrm{m}} f)(n) & = & [f(n)_1, f(n)_2, \ldots, f(n)_{|f(n)|}, \\ & & \quad 2f(n)_1, 2f(n)_2, \ldots, 2f(n)_{|f(n)|}] \end{array}$$

where $f(n)_i$ denotes the ith element in the list returned by $f(n)$ and $|f(n)|$ denotes its length. A function which returns the empty list can be used to "swallow" values:

$$\begin{array}{rcl} (\text{remove-odd} \circ_{\mathrm{m}} \mathrm{double})(3) & = & [6] \\ (\text{remove-odd} \circ_{\mathrm{m}} \mathrm{double})(4) & = & [4, 8] \\ (\text{remove-odd} \circ_{\mathrm{m}} \mathrm{clamp} \circ_{\mathrm{m}} \mathrm{double})(4) & = & [4] \end{array}$$

```
var coprimes = from x in Enumerable.Range( 1, n )
               where GCD( x, y ) == 1
               select x;
```

Listing B.1: Linq

Example B.2.2. *List comprehensions (Haskell, Python, C♯'s LINQ, . . .) are actually list monads in disguise. Listing B.1 can be written as*

$$\begin{array}{rcl}
\text{pack}(x) & = & [x] \\
\text{filter}(x) & = & \textbf{if } gcd(x,y) = 1 \textbf{ then } [x] \textbf{ else } [] \\
\text{range}(n) & = & [1, 2, \ldots, n] \\
\text{coprimes} & = & \text{pack} \circ_{\text{m}} \text{filter} \circ_{\text{m}} \text{range}
\end{array}$$

The list monad can also be interpreted as nondeterministic computing: every step generates multiple results, and further computation continues on each of these values. The list monad is extensively used in the formalisation of Featherweight VeriFast as a means to model nondeterminism.

B.3 The Maybe Monad

Null pointers (or references) are the bane of many programmers. While it is true that allowing pointers to be null gives us a simple means to express failure or a "lack of result", we generally do not wish to make use of this possibility. Exceptions are a more robust way of signaling failures and null objects are a cleaner way to represent "nonexistence". Unfortunately, many languages force us to deal with these edge cases where pointers are null, requiring special handling. It is generally not possible to express through the type system that a pointer is definitely not null, thus deferring the check at runtime: good practice dictates that we always check for null before accessing objects through pointers.

Many functional languages such as Haskell, O'Caml and Coq take the opposite approach: types normally do not allow null by default, except if explicitly asked. For this purpose, the Maybe type constructor is offered (also known as "option"). For example, in Haskell, a value of type [Int] is guaranteed to be a list, whereas in Java, it could also be null. If we wish to allow a special value indicating "no list", we can use Maybe [Int]. A list $[1,2,3]$ is then be written "Just $[1,2,3]$", and a lack of list is denoted "Nothing", which corresponds to null.

We now examine how we can compose functions with type signature

$$f : a \to \text{Maybe}\ b \qquad g : a \to \text{Maybe}\ b$$

We can interpret the result type as expressing the possibility of failure. If f returns a success value "Just x", we can just apply it to g. If, however, f fails, i.e. if it returns "Nothing", then the composition $g \circ_{\text{m}} f$ should also fail. One can view the Maybe monad as a (very simple) exception system, where execution automatically short-circuits after the first failure. Given this insight, it is easy to define the monadic composition:

$$g \circ_{\text{m}} f = \lambda\ x.\ \begin{array}{lll} \textbf{match}\ f(x)\ \textbf{with} & & \\ \quad \text{Just}\ x & \to & g(x) \\ \quad \text{Nothing} & \to & \text{Nothing} \end{array}$$

B.4 Kleisli Triples

As explained before, the monadic composition combines a $a \to T\ b$ and a $b \to T\ c$ function into a $a \to T\ c$ function. The most interesting aspect of this monadic composition is how it applies the second function on the results of the first, i.e. how to apply something of type $T\ b$ on a function with signature $b \to T\ c$. We can extract this core part:

$$- \gg\!= - \quad : \quad T\ a \to (a \to T\ b) \to T\ b$$

The monadic composition can then defined as

$$g \circ_{\text{m}} f = \lambda\ x.\ f\ x \gg\!= g$$

Another alternative is to define a lifting operator:

$$-^{\Uparrow} : (a \to T\ b) \to (T\ a \to T\ b)$$

We can express binding in terms of lifting:

$$x \gg\!= f \ = \ f^{\Uparrow}\ x$$

A monad's full definition consists of three components, called the Kleisli triple[3]:

- The type constructor T. The previous sections have examined the cases where $T\ a = [a]$ and $T\ a = \text{Maybe}\ a$.

[3]In order to form a valid Kleisli triple, the components also need to obey certain laws, but this falls out of the scope of this chapter.

- A way to compose functions of type $a \to T\ b$. Whether it takes the form of monadic composition, binding of lifting does not matter. For example, Haskell's monad type class requires a definition for binding.
- A unit function with type signature $a \to T\ a$. It defines a way to inject a value of an arbitrary type into a "monadic box". Following Haskell terminology, we will name this function return.

In the case of the List monad, the return function is defined as

$$\text{return}\ x = [x]$$

For the Maybe monad, it is

$$\text{return}\ x = \text{Just}\ x$$

B.5 Do Notation

While it is possible to implement monads in any language, it can quickly become a syntactic mess, undermining the monad's usefulness. To prevent this, languages can provide syntactic sugar to keep the code readable. Listing B.1 gives an example of C$^\sharp$'s syntax for monads.

During the formalisation of Featherweight VeriFast, we adopt the following notation which is heavily inspired on Haskell's do notation:

$$\begin{array}{rcl} \textbf{do}\ \begin{array}{l} x \leftarrow f \\ \mathit{rest} \end{array} & \equiv & f \gg\!= \lambda\ x.\ \mathit{rest} \\[1em] \textbf{do}\ \begin{array}{l} f \\ \mathit{rest} \end{array} & \equiv & f \gg\!= \lambda\ \_.\ \mathit{rest} \\[1em] \textbf{do}\ f & \equiv & f \end{array}$$

We can rewrite Listing B.1 as follows

$$\textbf{do}\ \begin{array}{l} x \leftarrow \text{range}\ n \\ \text{check}\ (\text{gcd}\ x\ y = 1) \\ \text{return}\ x \end{array}$$

where

$$\text{check}\ b = \lambda\ x.\ \textbf{if}\ b\ \textbf{then}\ [x]\ \textbf{else}\ []$$

The first line picks a x in the given range; the rest of the computation will be performed for every x in $[1, 2, \ldots, n]$. The second line filters out cases where x and y are not coprime. The last line states that x is the result of the computation. Thus, the end result will be a list of all x between 1 and 10 which are coprime with y. If $y = 12$, the result would be $[1, 5, 7]$.

Used with the Maybe monad, we get:

$$\begin{aligned}\textbf{do}\ & a \leftarrow f \\ & b \leftarrow g\ a \\ & c \leftarrow h\ a\ b \\ & \text{return}\ (a, b, c)\end{aligned}$$

If f succeeds, a will be bound to its value. We then compute $g\ a$ and save its value in b, etc. Only if all three functions f, g and h succeed do we reach the final line, which returns the three results in a tuple. Note that the do notation has the advantage of considerably simplifying access to the values of a, b and c. Expressing the same computation without do notation requires more explicit handling of these values, leading to poorly readable code.

B.6 The State Monad

Haskell is a purely functional language: it does not allow destructive modifications, thus being completely stateless. A consequence is that functions will always return the same result, given the same arguments. This is not always desirable: for example, performing IO inherently depends on something external. A pseudorandom number generator (PRNG) would also be useless if it always returned the same value; it needs some own "private local state" in order to function correctly. In this section, we examine a way to solve this problem.

Let us consider some stateful function f with type signature $a \to b$. Its statefulness means it is allowed to read from and write to the current state (e.g. the heap). State can be viewed as an implicit argument and secundary return value: we can turn f into a stateless function by making state explicit:

$$f : (a \times state) \to (b \times state)$$

Thus, the function receives the current state as argument and returns an updated version of it. Even though we can use regular function composition to chain such functions together, it does not help us much in practice, as it is seldom the case that our code perfectly fits this "relay race mold": for example, the result of one function could need to be applied to more than one function. While it is certainly possible to accomplish this, it quickly becomes syntactically burdensome.

We can take advantage of the do notation if we can formulate state as a monad. We will show that we can make state implicit again, having built an abstraction layer which effectively builds a stateful domain specific language on a stateless foundation. Assuming S is the state type, we can define the State type constructor as

$$T\ a = S \to a \times S$$

The bind operation then becomes

$$\begin{array}{rcl} - \gg\!= - & : & \overbrace{(S \to a \times S)}^{T\ a} \to \overbrace{(a \to S \to b \times S)}^{a \to T\ b} \to \overbrace{(S \to S \times b)}^{T\ b} \\ f \gg\!= g & = & \lambda\ s_0.\ \textbf{let}\ (a, s_1)\ =\ f\ s_0\ \textbf{in}\ g\ a\ s_1 \end{array}$$

With some deciphering, it can be seen that f first transforms the state into s_1, which is then fed as the "current state" to g. Also, f's return value is passed to g along with the state, as one would expect from regular function composition.

We have define the type constructor and the binding operation, only the unit function remains:

$$\text{return}\ x = \lambda\ s.\ (x, s)$$

We now define two helper functions:

$$\begin{array}{rcl} \text{get-state} & = & \lambda\ s.\ (s, s) \\ \text{set-state}\ s' & = & \lambda\ s.\ (\square, s') \end{array}$$

The function get-state returns the current state (first item of the returned pair), and keeps the state unchanged (second item of the returned pair). To modify the current state to a new state s', we use set-state(s'). The return value is $\square$, similar to **void** in other languages.

The State monad is relied upon for the formalisation of Featherweight VeriFast to model the execution of SIL. In a similar vein, we apply the State monad in defining an RPN calculator as an illustration.

Example B.6.1. *Reverse Polish notation (RPN) is a postfix mathematical notation: $a + b$ is written $a\ b\ +$, etc. We view the code as a list of tokens which are either numbers of operators. Evaluation consists of having a stack and "execute" each token in turn, i.e. let it affect the stack in some way. For example, evaluating* $5\ 3\ +\ 2\ \times$ *in a stepwise manner gives*

$$[] \xrightarrow{5} [5] \xrightarrow{3} [3,5] \xrightarrow{+} [8] \xrightarrow{2} [2,8] \xrightarrow{\times} [16]$$

```
push(n)          =  do s ← get-state
                       set-state (n : s)

pop              =  do s ← get-state
                       let top = head s in
                       set-state (tail s)
                       return top

step(n)          =  do push n
step(+)          =  do x ← pop
                       y ← pop
                       push (x + y)
                 ⋮

execute []       =  do return □
execute (t : ts) =  do step t
                       execute ts
```

Figure B.1: RPN Calculator

Let us formalise this execution using the State monad. The state has type [int]*, i.e. a list of integers. We can then define an execution function as shown in Fig. B.1.*

The functions push *and* pop *build an abstraction layer upon* get-state *and* set-state*:* push(n) *adds n to the stack,* pop *removes the top of the stack and returns it. The* step *function handles two cases:*

- *Numbers are simply pushed on the stack.*
- *Evaluating the + operator pops two values from the stack and pushes their sum back on.*

Lastly, the execute *function executes all tokens in turn. Notice how the code looks imperative, even though we work in a purely functional setting.*

In this example, we have conveniently ignored what happens when the stack does not contain enough values, e.g. what happens we evaluate "1 +"? When such a situation arises, execution should fail and further tokens should be ignored. This strongly reminds us of the Maybe monad discussed in Sect. B.3.

B.7 Combining the State and Maybe Monads

The previous section ended with a climax which needs to be resolved. On the one hand, we have the State monad which allows us to elegantly deal with state, while on the other hand we can use the Maybe monad to automatically take care of failure. In this section, we show how to combine both monads.

To support failure, we revise the Kleisli triple:

$$\begin{array}{rcl}
T\,a & = & S \to \text{Maybe}\ (a \times S) \\[1em]
f \gg\!= g & = & \lambda\, s_0.\ \ \textbf{match}\ f(s_0)\ \textbf{with} \\
 & & \qquad \begin{array}{lcl} \text{Just}\ (a, s_1) & \to & g(a, s_1) \\ \text{Nothing} & \to & \text{Nothing} \end{array} \\[1em]
\text{return}(x) & = & \lambda\, s.\ \text{Just}\ (\Box, s)
\end{array}$$

The type constructor has been extended with a Maybe, allowing us to use Just(a, s) and Nothing to represent successful execution and failure, respectively. Binding needs to deal appropriately with failure: if the first result yields Nothing, the entire operation fails. In case of success, we proceed in the same way as defined in the previous section.

The helper functions get-state and set-state require a small update. We also add a third helper function:

$$\begin{array}{rcl}
\text{get-state} & = & \lambda\, s.\ \text{Just}\ (s, s) \\
\text{set-state}\ s' & = & \lambda\, s.\ \text{Just}\ (\Box, s') \\
\text{fail} & = & \lambda\, s.\ \text{Nothing}
\end{array}$$

The lower abstraction level has now been adapted to accommodate failure. The only RPN specific function (Fig. B.1) which requires an update is pop:

$$\begin{array}{lll}
\text{pop} = & \textbf{do} & s \leftarrow \text{get-state} \\
 & & \textbf{match}\ s\ \textbf{with} \\
 & & \qquad \begin{array}{lcll} [\,] & \to & \textbf{do} & \text{fail} \\ n : ns & \to & \textbf{do} & \text{set-state}\ ns \\ & & & \text{return}\ n \end{array}
\end{array}$$

i.e. pop fetches the current stack and checks if it is empty. If so, pop fails. If not, it removes the top item and returns it.

B.8 Combining the State and List monads

As a last example, we show how to combine the State and List monad. We use this to simulate nondeterministic execution in our formalisation of Featherweight VeriFast.

Instead of working with single integers, we generalise our language to work with ranges, from which the nondeterminism arises. For example, we want the program $[1 \ldots 4]$ $[4 \ldots 5]$+ to compute $x + y$ for all x ranging from 1 to 4 and y ranging from 4 to 5. Thus, the expected result is $[5,6,6,7,7,8,8,9]$. Since we work we lists, duplicate values can appear. We can interpret this extra information as probabilities. If we are only interested in the actual values, we could either work with sets instead of lists, or let the binding operation remove duplicates from the list.

The Kleisli triple becomes

$$\begin{aligned}
T\ a &= S \rightarrow [a \times S] \\[1em]
f \ggg g &= \quad \text{concat} \circ \text{map}\ g' \circ f \\
&\qquad\quad \text{where } g'(a,s) = g\ a\ s \\[1em]
\text{return}\ x &= \lambda\ s.\ [(\square, s)]
\end{aligned}$$

We modify the helper functions accordingly:

$$\begin{aligned}
\text{get-state} &= \lambda\ s.\ [(s,s)] \\
\text{set-state}\ s' &= \lambda\ s.\ [(\square, s')] \\
\text{fail} &= \lambda\ s.\ [] \\
\text{choice}\ xs &= \lambda\ s.\ \text{map}\ (\lambda\ x.\ (x,s))\ xs
\end{aligned}$$

The choice function is the primitive to be used to fork execution into multiple forks: for every x in xs a new path will be generated. The function returns a different value x for each path. We also still provide some sort of failure, represented by the empty list. For example, executing $[1 \ldots 4]$ + yields $[]$. An updated version of the RPN calculator is shown in Fig. B.2.

```
        push(n)  =  do  s ← get-state
                        set-state (n : s)

            pop  =  do  s ← get-state
                        match s with
                           []      →   do  fail
                           n : ns  →   do  set-state ns
                                           return n

       step(xs)  =  do  n ← choice xs
                        push n

        step(+)  =  do  x ← pop
                        y ← pop
                        push (x + y)

        step(−)  =  do  x ← pop
                        y ← pop
                        push (x − y)

        step(×)  =  do  x ← pop
                        y ← pop
                        push (x × y)

                 ⋮

     execute []  =  do  return □
execute (t : ts) =  do  step t
                        execute ts
```

Figure B.2: Nondeterministic RPN Calculator

Appendix C

(An Attempt At) A Short Introduction to Coq

In this chapter, we introduce the user to the basics of Coq. Some sections discuss specific parts of the thesis. It is not our intention to be either complete or a hundred percent accurate: given the complexity of Coq it would take too many pages to achieve this. Our intention is only to give the reader who is unfamiliar with Coq an intuitive understanding of the Coq scripts included in this text. For more complete (and exact) information, we refer the reader to Coq'Art [16], Chlipala's Certified Programming with Dependent Types [29], or Pierce's Software Foundations [96].

C.1 The Curry-Howard Isomorphism

At the heart of Coq lies the Curry-Howard isomorphism. In short, the Curry-Howard isomorphism [65] states there is a correspondence between values and proofs, and between types and propositions.

$$\begin{array}{rcl} \text{types} & \Longleftrightarrow & \text{propositions} \\ \text{values} & \Longleftrightarrow & \text{proofs} \end{array}$$

Let us agree on the definitions[1] of the following four words we used in this definition:

Value A value is a piece of data. The integer `5` is an example, as is `true`. Objects in object oriented languages are all values too.

Type A value always has a certain type. The type of a value is the set of values to which it belongs. For example, `int` is the type of the value `5` and `bool` is the type of `true`. In object oriented languages, the type of an object is its class.

Proposition A proposition is a logical formula which can be either true or false. For example, $3 \leq 5$ is a proposition, and so is $5 \leq 3$.

Proof A proposition is true if it is possible to construct a proof for it. While we already know that $3 \leq 5$, we must technically construct a proof for it to be sure. We also know that $5 \leq 3$ is false, which means that is is impossible to prove it[2].

Admittedly, to most readers, this is a rather unhelpful explanation. To illustrate the concept, we will show in the following sections how we can we can create a list class in C++ for which we statically check that the indexing operation never goes out of bounds, i.e. the program will not compile if we cannot prove the index is valid. In a next step, we will relate this to Coq.

C.2 Curry-Howard in C++

In this section, we illustrate the Curry-Howard isomorphism by means of examples in C++. This code relies heavily on templates, which the reader might be unfamiliar with. With this in mind, we have attempted to explain all examples as best we could.

C.2.1 Objects as Witnesses

In object oriented languages, we often use class invariants to define what states an object is allowed to be in. For example, a file name must obey some OS specific rules: Windows does not allow file names to contain any of

[1]Warning: the expert reader might bawl at the sight of some of these definitions.

[2]To make sure $5 \leq 3$ is indeed false, we should construct a proof that shows that $5 \leq 3$ leads to a contradiction.

the following characters: `/ \ : * ? " < > |`. It is possible to define a class `FileName` that enforces these rules, i.e. only `FileName` objects that represent a valid file name are allowed to exist. This can easily be achieved by performing the necessary checks in the constructor, as is shown in Listing C.1.

A little clarification for readers unfamiliar with C++: the `ValidatedFileName` class (lines 3–16) takes a compile-time argument `VALIDATOR` which is used to determine the validity of a file name. In other words, `VALIDATOR` essentially represents a predicate. An example of such a validator is the class `WindowsValidator` (lines 18–26). Line 10 is responsible for checking the file name: a `VALIDATOR` object is created on the stack meaning its constructor is called with the file name to be checked. The `VALIDATOR` constructor is expected to throw an exception if the given file name is invalid, so that the construction of the `ValidatedFileName` object gets aborted. Line 28 defines the type `FileName` which is merely `ValidatedFileName` parameterized with `WindowsValidator`. When executing the program, the construction of `fn1` on line 31 will succeed, but an exception will be thrown while executing line 32.

The `ValidatedFileName` example is an illustration of how the existence of an object can have a meaning: if one gets their hands on a `FileName` object, one knows for certain that the file name it holds is valid. In other words, the mere existence of the `FileName` object implies the validity of its contents.

C.2.2 Moving it to Compile-Time

In the example shown in Listing C.1, the validity checks are done at runtime. Say we have a program making use of a large number of such hard coded file names. In order to make sure all these file names are correct, we would have to run the program and make sure all paths on which `FileName` objects are created are visited at least once. If no exception is thrown during this execution, we know all of these hard coded file names are valid.

This is a rather clumsy approach; we feel a better solution should be possible. The compiler has access to all hard coded file names, i.e. it has all necessary information on hand, so we should be able to ask it to check all of them while it compiles the source. While it is technically possible to implement this in C++, the result would be hard to understand and thus make for a very poor illustration of the concept we are trying to explain.

For this reason, we switch to a much more lightweight but more academic example: whereas our previous example used `FileName` objects as witnesses to the validity of their contents, we now introduce the class `LEQ` with two fields `X` and `Y` whose objects are only allowed to exist if the `X`-field contains a value less

```cpp
#include <string>

template<class VALIDATOR> class ValidatedFileName {
private:
  std::string _filename;

public:
  ValidatedFileName(const std::string& filename)
      : _filename(filename) {
    VALIDATOR validator(filename);
  }

  std::string as_string() const {
    return _filename;
  }
};

class WindowsValidator {
public:
  WindowsValidator(const std::string& filename) {
    if ( filename.find_first_of("/\\:*?\"<>|") !=
          std::string::npos ) {
      throw "invalid-filename";
    }
  }
};

typedef ValidatedFileName<WindowsValidator> FileName;

int main() {
  FileName fn1("abc.txt");   // ok
  FileName fn2("ab:c.txt");  // fails
}
```

Listing C.1: File Names in C++

```
class LEQ {
private:
  int _x;
  int _y;

public:
  LEQ(int x, int y) : _x(x), _y(y) {
    if ( x > y )
      throw "invalid-arguments";
  }
};
```

Listing C.2: `LEQ` in C++

than or equal than the value stored in the `Y`-field. A possible implementation is shown in Listing C.2.

In order to determine whether $3 \leq 5$, we can now write a program containing `LEQ(3, 5)` and see if it crashes, i.e. the actual checking occurs at runtime. We can improve on this by moving the check up to compile-time. In other words, we want to write a program which only compiles on condition that $3 \leq 5$.

To achieve this, we rewrite our `LEQ` class. In a first step, we must upgrade the fields `x` and `y` to compile-time variables, i.e. we introduce two template parameters `X` and `Y`. We now want `LEQ<X, Y>` objects only to exist if `X <= Y`. One way of doing this is, like before, throwing an exception in the constructor if `X > Y`, but this would be a runtime check instead of a compile-time one.

Before we define the `LEQ<X, Y>` class, we first have to define what $\leq$ means[3]. We define the relation $\leq$ inductively by means of the following two rules:

$$\frac{}{x \leq x}\ \leq\text{-}\textsc{refl} \qquad \frac{x \leq y}{x \leq y+1}\ \leq\text{-}\textsc{succ}$$

For example, the following derivation tree shows that $3 \leq 5$:

$$\dfrac{\dfrac{\dfrac{}{3 \leq 3}\ \leq\text{-}\textsc{refl}}{3 \leq 4}\ \leq\text{-}\textsc{succ}}{3 \leq 5}\ \leq\text{-}\textsc{succ}$$

It is clear that this definition of $\leq$ yields the same results as our intuitive notion of the order relation: to prove $x \leq y$ where x is indeed less than or equal to y,

[3] While it is possible to just use C++'s built-in <= operator, we prefer to define our own as a proof of concept that we can define our own custom relations.

```
 1 template<unsigned X, unsigned Y>
 2 class LEQ;
 3
 4 template<unsigned X>
 5 class LEQ<X, X> { };
 6
 7 template<unsigned X, unsigned Y>
 8 class LEQ {
 9 public:
10   LEQ(LEQ<X, Y-1> proof) { }
11 };
```

Listing C.3: `LEQ<X, Y>` in C++

we start with $x \leq x$ (using ≤-refl) and then increment the right hand side until it reaches y using ≤-succ.

The fact that there are two rules (≤-refl and ≤-succ) means there should be two ways to create a `LEQ<X, Y>` object, i.e. there should be two ways to construct `LEQ<X, Y>` objects. The C++ code is shown in Listing C.3.

A little bit of clarification may be in order:

- Lines 1–2 declare the `LEQ<X, Y>` class: the two compile-time variables `X` and `Y` are (unsigned) integers. Note the absence of a class body: this means the class is only *declared* but not yet *defined*. This declaration alone does not allow us to instantiate any `LEQ<X, Y>` objects.
- Lines 4–5 define the first "constructor": it allows any object `LEQ<X, X>` to be constructed, i.e. it corresponds to the ≤-refl rule. Notice the class body: it is present but empty. In such a situation, similarly to Java and C♯, C++ automatically provides us with a default constructor, i.e. a parameterless one.
- Lines 7–11 define the second "constructor" and thus corresponds to the ≤-succ rule. The constructor on line 10 is of particular interest: in order to build a `LEQ<X, Y>` object, the programmer is required to provide a `LEQ<X, Y-1>` object (referred to as `proof` in the code) first.

Listing C.4 shows how to proceed to prove that $3 \leq 5$, or, in other words, how to construct a `LEQ<3, 5>` object. The three lines of code correspond exactly to the three steps in the derivation tree shown on page 255:

```
1 LEQ<3, 3> leq33;
2 LEQ<3, 4> leq34(leq33);
3 LEQ<3, 5> leq35(leq34);
```

Listing C.4: Proving that $3 \leq 5$ in C++

- On line 1, we first use ≤-REFL to prove that $3 \leq 3$.
- On line 2, we rely on ≤-SUCC to prove that $3 \leq 4$. Note how we pass along the proof object `leq33` to the constructor so that it may build an object of type `LEQ<3, 4>`, which we name `leq34`.
- The construction of `leq35` is similar. Its type is `LEQ<3, 5>`, which means we have just proved that $3 \leq 5$.

Given the definition in Listing C.3, we are now able to let the compiler check our proof trees. Listing C.4 actually builds a proof that $3 \leq 5$ and it is fully checked at compile time, meaning there is no need to execute the program.

C.2.3 Compile-time Checked Indexing

We are now ready to create a list data structure for which the indices are checked at compile time. The code is shown in Listing C.5. We clarify:

- Line 1 states that a list has two compile-time arguments: `T` is type of the list items and is comparable to the type parameter in generics in Java and C$\sharp$. The `LEN` template parameter represents the length of the list. For example, `list<int, 3>` is the type of lists containing three integers.
- Line 4 declares the `_items` fields, i.e. an array of `LEN` items of type `T`.
- Lines 7–10 defines the `at` method and allows us to fetch the element with index `I`. This index `I` is declared as a compile-time variable (line 7) and is used on line 9 to index the array `_items`. Note that `at` takes one parameter of type `LEQ<I, LEN-1>`. This effectively means that it first requires a proof that `I` is a valid index before it will attempt to reach into the array `_items`. Also note that we haven't even named this parameter as we do not need to manipulate the actual proof object: we only demand that it exists.

```
template<class T, unsigned LEN>
class list {
private:
    T _items[LEN];

public:
    template<unsigned I>
    T& at(LEQ<I, LEN-1>) {
        return _items[I];
    }
};
```

Listing C.5: A List with Compile-Time Index Checking

```
1  int main() {
2    LEQ<3, 3> leq33;
3    LEQ<3, 4> leq34(leq33);
4    LEQ<3, 5> leq35(leq34);
5
6    list<int, 6> lst;
7    lst.at<3>(leq35) = 2;
8    int x = lst.at<3>(leq35);
9  }
```

Listing C.6: Example Usage of a Compile-Time Checked List

Listing C.6 shows how we can make use of this list class. On line 6 we create an integer list of length 6, after which (line 7) we assign[4] 2 to the item with index 3. In order to do this, we pass along the proof that 3 is a valid index for a list of length 6, i.e. that $3 \leq 5$. On line 8, we retrieve this value again and store it in `x`. Again a proof object representing the validity of 3 as an index is required.

C.2.4 Disadvantages

We have shown how we could introduce compile-time checks against index out of bounds errors in C++. However, there are multiple serious disadvantages:

[4]We can use `at` for this purpose as this method returns a *reference* to the item.

- All indices must be compile-time variables, meaning that their value must not depend on any data collected at runtime. For example, we cannot have a list of items from which the user can pick one during program execution, as his or her choice is not available at compile time. Similarly, the length of a list must be fixed at compile time.
- If we need a list which supports indexing with a runtime index, we need to reimplement it separately. This new version will at best have runtime index checking. We would have two version of each class: one with and one without compile-time checking.
- Related to the previous point is the need to reinvent many basic things, such as conditionals and loops, to make them compatible with compile-time variables. Listing C.7 demonstrates how a compile-time loop can be implemented in C++.
- It is possible to create arbitrary proof objects. Listing C.8 shows how we can abuse casts to circumvent the type system: C++ does not allow us to cast directly to an arbitrary `LEQ<X, Y>`, but through the use of pointers it is possible to pretend one has proven arbitrary propositions. This more or less corresponds to axiomatically stating new facts, which can compromise the soundness of the whole system.
- In order to work with compile-time variables, one needs to make extensive use of templates, which were originally not intended to be used for such advanced purposes. All kinds of trickery are necessary, especially when considering the limitations C++ imposes (e.g. no partial template specialisation).

We have shown how proofs can be encoded as objects in C++, thereby illustrating the Curry-Howard isomorphism: `LEQ<X, Y>` is a type, but it also corresponds to the proposition $x \leq y$. An object of type `LEQ<X, Y>` represents a proof of $x \leq y$. Next, we will show how we can implement the same in Coq without having to deal with all these shortcomings.

C.3 Curry-Howard in Coq

We revisit the examples from the previous sections, but implement them in Coq.

```
template<unsigned N,
         template<unsigned A> class F,
         unsigned K = 0,
         unsigned STEP = 1>
struct Repeat
{
    Repeat()
    {
        F<K>();
        Repeat<N, F, K + STEP, STEP>();
    }
};

template<template<unsigned A> class F,
         unsigned K,
         unsigned STEP>
struct Repeat<K, F, K, STEP>
{
    Repeat() { }
};

template<unsigned N>
struct Print
{
    Print()
    {
        std::cout << N << std::endl;
    }
};

int main()
{
    Repeat<10, Print>();
}
```

Listing C.7: A Loop Using Templates

```
template<unsigned X, unsigned Y>
LEQ<X, Y> cheat() {
  LEQ<0, 0>  leq;
  LEQ<0, 0>* p   = &leq;
  LEQ<X, Y>* pxy = reinterpret_cast<LEQ<X, Y>*>(p);
  LEQ<X, Y>  lxy = *pxy;

  return lxy;
}
```

Listing C.8: Circumventing C++'s Type System

C.3.1 Implementing ≤

We now define ≤ in Coq. As a reminder, we repeat the two derivation rules:

$$\frac{}{x \leq x}\ \leq\text{-REFL} \qquad \frac{x \leq y}{x \leq y+1}\ \leq\text{-SUCC}$$

Listing C.9 shows how to implement this in Coq. We clarify:

- `leq` is our new type that corresponds to ≤. The type is parameterized in two natural numbers (`nat`), meaning that the type `leq 3 5` corresponds to the proposition $3 \leq 5$. `Prop` is short for proposition; we will discuss `Prop` in more detail later.
- `leq` is an *inductive type* with two constructors, which define the only two building blocks to construct a proof of $x \leq y$.
- `leq_refl` can be seen as a function taking one argument `n` of type `nat` and returning a proof of $n \leq n$.
- `leq_succ` takes three arguments: two natural numbers `n` and `m`, and a proof that $n \leq m$. Given these, it returns a proof that $n \leq m + 1$.

Notice how the two constructors correspond to the two definitions for `LEQ` in Listing C.3.

Listing C.10 shows how to proceed to prove that $3 \leq 5$: `leq_33` is a value of type `leq 3 3` and thus also a proof of $3 \leq 3$. It is built using the `leq_refl` constructor. In the next steps, we build proofs of $3 \leq 4$ and $3 \leq 5$ using the `leq_succ` constructor.

```
Inductive leq : nat -> nat -> Prop :=
  | leq_refl : forall n : nat, leq n n
  | leq_succ : forall n m : nat, leq n m -> leq n (S m).
```

Listing C.9: ≤ in Coq

```
Definition leq_33 : leq 3 3 :=
  leq_refl 3.

Definition leq_34 : leq 3 4 :=
  leq_succ 3 3 leq_33.

Definition leq_35 : leq 3 5 :=
  leq_succ 3 4 leq_34.
```

Listing C.10: 3 ≤ 5 in Coq

C.3.2 Implementing Lists

We now turn our attention to lists. Whereas we used arrays to represent lists in C++, we will now implement them as linked lists. Listing C.11 shows the Coq code.

- Lists are defined as an *inductive data type*. Note how the structure of a list is similar to that of `leq` which we defined in the previous section.
- The `List` type takes two arguments: the first one (explicitly named `A`) is the type of the items. The second (nameless) parameter has type `nat` and represents the length of the list. For example, `list nat 8` is the type of lists containing eight natural numbers.
- The first constructor, `cons`, represents a "cons cell", which is a building block of (linked) lists. Each cons cell contains one value (the *head* of the list) and a link to another, smaller list (the *tail* of the list). `cons` takes four arguments:
  - An item type `A`. This parameter comes from the first line (`A : Set`).
  - A natural number `n` which denotes the length of the tail.
  - The head of type `A`.
  - The tail of type `list A n`.

```
Inductive list (A : Set) : nat -> Set :=
 | cons : forall (n : nat),
            A -> list A n -> list A (S n)
 | nil  : list A O.
```

Listing C.11: List in Coq

These four together form a new list of type `list A (S n)`.

- The second constructor, `nil`, takes a single argument: the list item type `A`. `nil A` represents the empty list and has type `list A 0`.

For example, the list $[1,2,3]$ is written as follows:

```
cons nat 2 1 (cons nat 1 2 (cons nat 0 3 (nil nat)))
```

C.3.3 Implementing List Indexing

Defining the actual indexing operation is much more complex; we will limit ourselves to a very brief explanation. Listing C.12 shows how it can be implemented in Coq. We left out some definitions as they do not offer any extra insights.

- Lines 1–4 contain a first auxiliary definition. `list_head` takes a list `lst` of length `S n` (meaning it is at least one item long) and returns its head.
- Lines 6–9 define `list_tail`, which returns the tail of the given list `lst`.
- Lines 11–12 prove that $n + 1 \leq m \Rightarrow n \leq m$.
- Lines 14–24 define the actual list indexing operation `list_at` which takes five arguments:
  - `A` denotes the type of the list items.
  - `n` must be introduced to be able to denote the length of the list in the following argument.
  - `lst` is a list of length `S n` as indexing on a zero length list is not allowed.
  - `i` is the index.
  - `valid` is a proof that `i` is a valid index.

- The actual definition on lines 19–23 shows an example of the tactic language in Coq. It allows us to input definitions in an interactive way.
  - Line 19 tells Coq we wish to take a look at the index `i`, considering the cases where `i = 0` and `i = S i'` separately.
  - First, Coq asks us to deal with the case `i = 0`. Line 20 responds that we just take the head of the list.
  - Now Coq presents us with the case `i = S i'`, i.e. $i' = i - 1$. Dealing with this requires multiple steps (lines 21–23).
  - First, on line 21, we tell Coq to take the tail of the list and call it `tail`.
  - We want to call `list_at` recursively: we know that $xs[i] = \text{tail}(xs)[i']$. To be able to make this recursive call, we need a proof that `i` is a valid index. We already know that `leq (S i) n` (this is automatically provided by Coq, it follows from the induction hypothesis), so using `leq_Sn_m` we can derive `leq i n` from it. We do this on line 22 and name the proof `valid_i'`.
  - Line 23 makes the recursive call using this proof.

  Without making use of the tactic language, the code for `list_at` would look like Listing C.13. Explaining it falls far outside the scope of this introduction.

C.4 Type Hierarchy

In the previous sections, we have discussed the Curry-Howard isomorphism, which states that we can represent proofs by objects. Thus, we have two kinds of objects: those that represent regular data structures (e.g. natural numbers, lists, . . .), and those that represent proofs (e.g. a proof of $3 \leq 5$).

Coq's type system makes the distinction between the two by assigning different metatypes to data and proof objects. Figure C.1 gives an overview. As shown, the values 0, 1, 2, etc. have type `nat`. `nat` itself is also a first class citizen in Coq (i.e. it can be passed along as argument or returned as a result) and its type is `Set`. `Set` is also a first class citizen, and its type is `Type(0)`. This goes on forever: `Type(0)` has type `Type(1)`, etc.

On the same metalevel as `Set` resides `Prop`, which is the type of all propositions. Examples are `leq 3 5` and `leq 5 3`. One metalevel below propositions we find proof objects. For example, `leq_35` (see Listing C.10) has type `leq 3 5` and thus represents a proof of $3 \leq 5$. Since $5 \leq 3$ is false, the type `leq 5 3`

```
Definition list_head {A : Set}
                     {n : nat}
                     (lst : list A (S n)) : A.
(* ... *) Defined.

Definition list_tail {A : Set}
                     {n : nat}
                     (lst : list A (S n)) : list A n.
(* ... *) Defined.

Lemma leq_Sn_m :
  forall n m : nat, leq (S n) m -> leq n m.
Proof. (* ... *) Qed.

Definition list_at {A : Set}
                   {n : nat}
                   (lst : list A (S n))
                   (i : nat)
                   (valid : leq i n) : A.
  induction i as [ | n' rec ].
  apply (list_head lst).
  assert (tail := list_tail lst).
  assert (valid_i' := leq_Sn_m _ _ valid).
  apply (rec valid_i').
Defined.
```

Listing C.12: List indexing in Coq (Ltac)

```
list_at =
  fun (A : Set) (n : nat) (lst : list A (S n))
      (i : nat) (valid : leq i n) =>
    let H := nat_rec
               (fun i0 : nat => leq i0 n -> A)
               (fun _ : leq 0 n => list_head lst)
               (fun (i' : nat) (rec : leq i' n -> A)
                    (valid0 : leq (S i') n) =>
                  let tail := list_tail lst in
                  let valid_i' := leq_Sn_m i' n valid0 in
                  rec valid_i') i
    in H valid
```

Listing C.13: List indexing in Coq (Gallina)

⋮					
Type(2)					
Type(1)					
Type(0)					
Set			Prop		
nat	bool	⋯	leq 3 5	leq 5 3	⋯
0 1 2	true false	*data*	leq_35	-	*proofs*

Figure C.1: Coq Metatype Hierarchy

```
Axiom proof_irrelevance :
  forall (P : Prop) (p1 p2 : P), p1 = p2.
```

Listing C.14: List indexing in Coq (Gallina)

is uninhabited, meaning there exist no proofs of that proposition. Note how propositions and types reside on the same level, as do values and proofs. This is a result of the Curry-Howard isomorphism.

An interesting concept is that of proof irrelevance. A proposition can be proven in different ways, yielding different proof objects. Since it is not important how a certain proposition has been proven, all proof objects of the same type can be considered equal. This is exactly what the `proof_irrelevance` axiom in the Coq standard library states (see Listing C.14).

C.5 Extraction

Coq provides us with a purely functional programming language with a powerful type system, enabling us to express a wide variety of propositions. This allows us to write programs and prove full functional correctness. For example, Listing C.15 shows what the type of a sorting function (aptly named `sorting_function`) would look like in Coq.

- Lines 1–8 define what it means for a function $f : A \to B$ to be surjective, injective and bijective.
- For the sake of readability, we model lists with item type A as functions with signature $\mathbb{N} \to A$, i.e. we only support infinite lists. Working with

```
 1 Definition surjection {A B : Type}
 2                       (f : A -> B) : Prop :=
 3   forall b : B, exists a : A, f a = b.
 4
 5 Definition injection {A B : Type}
 6                      (f : A -> B) : Prop :=
 7   forall a a' : A, f a = f a' -> a = a'.
 8
 9 Definition bijection {A B : Type}
10                      (f : A -> B) : Prop :=
11   surjection f /\ injection f.
12
13 Definition permutation {A : Set}
14                        (f g : nat -> A) : Prop :=
15   exists p : nat -> nat,
16     bijection p /\ forall n : nat, f n = g (p n).
17
18 Definition in_order {A : Set}
19                     (before : A -> A -> bool)
20                     (f : nat -> A) : Prop :=
21   forall n : nat, before (f n) (f (S n)) = true.
22
23 Definition sorting_function
24   {A : Set}
25   (before : A -> A -> bool)
26   (f : nat -> A) : { g : nat -> A | permutation f g /\
27                                     in_order before g }.
```

Listing C.15: Type of Sorting Function

finite lists is certainly possible but would require much more elaborate syntax and explanations.

- On lines 10–12 we define the concept of a permutation on lists: we say that f and g are permutations of each other if there exists a bijection $p : \mathbb{N} \to \mathbb{N}$ such that $f(n) = g(p(n))$.
- Lines 14–17 define the `in_order` predicate, which expresses that elements in the list f are ordered according to the complete order described by `before`.
- The type of sorting functions is defined on lines 19-24. A sorting function is a function that given a type `A` of items, a complete order `before` and a

```
Definition sorting_function
  {A : Set}
  (before : A -> A -> bool)
  (f : list A) : { g : list A | permutation f g /\
                                in_order before g }.
```

Listing C.16: Type of Sorting Function (Finite Lists)

list `f` returns a new list `g` for which is guaranteed that it is a permutation of `f` (`permutation f g`) and that it is in order (`in_order before g`). The type `{ a : A | P a }` expresses that some value a of type 'A' is returned, together some guarantees `P a` in the form of proof objects.

Any function we define that has this type is a sorting function. In other words, Coq allows us to check that a certain function implements some sorting algorithm correctly. An unfortunate consequence of our concession of making use of infinite lists for the sake of readability is that it is impossible to define a function which has this type signature, as function have to be computable and merely finding the smallest element in an infinite list is impossible to do in finite time. We remedy this by giving a slightly adapted version of the definition for `sorting_function` in Listing C.16. We have left out the auxiliary definitions as they are rather complex.

Let us say now that we implement a function of the type `sorting_function`. We would have a sorting function which has been proven correct in our hands, but there are serious restrictions regarding its use: it can only run within the Coq environment, and it is far from efficient.

For this reason, Coq allows us to *extract* code, which in essence consists of translating it into another language (O'Caml or Haskell). During this process, only what influences the behaviour at runtime is preserved, meaning all proofs can be discarded. In our case, an extracted sort function would only return `g`, as the extra proofs are only needed to prove the correctness of the program, not for its succesful execution.

How does Coq know what it can safely leave out during extraction? In simple terms, Coq makes use of the distinction between `Prop` and `Set`: values that fall under `Prop` are not allowed to have any influence on the runtime semantics of a program. This is enforced by Coq's type system.

A simple example of this is disjunction. Say we have an algorithm which depends on the primality of its argument, which means we need to be able to determine whether some number is prime or not. The proposition

```
Theorem sumbool_impl_or :
  forall (P Q : Prop), {P} + {Q} -> P \/ Q.
Proof.
  intros P Q H; destruct H; [ left | right ]; trivial.
Qed.

Theorem or_impl_sumbool :
  forall (P Q : Prop), P \/ Q -> {P} + {Q}.
Proof.
  (* Impossible to prove *)
```

Listing C.17: `or` vs `sumbool` Type

```
forall n : nat, prime n \/ ~ prime n
```

states a natural number `n` is either prime or not prime, but a proof of this fact does not describe how one determines which case holds.

Coq provides second kind of disjunction, written

```
forall n : nat, { prime n } + { ~ prime n }
```

which resides in the `Set` world, thus blurring the division between `Prop` and `Set`, as proof object can also exist as `Set` objects. The difference with the `Prop` variant is that objects of this type also contain the algorithm to decide in a finite number of steps whether a number is prime or not. Thus, in an algorithm, one is permitted to use the `Set` disjunction `{ }+{ }` in a conditional statement, but not the `Prop` variant.

This might give the reader the impression that duplication is necessary: we would have to prove `P \/ Q` and if computable also `{ P }+{ Q }`. Fortunately, the former can easily be proven from the latter, as the latter is "strictly more powerful", as illustrated in Listing C.17. Thus, using `sumbool_impl_or` allows us to directly translate the `Set` variant into its `Prop` twin.

We can relate this to the distinction made between compile-time and runtime variables in C++: `Prop` more or less corresponds to compile-time proof objects, while the runtime world coincides with `Set`. While C++ requires the use of two different "sublanguages" (i.e. regular C++ such as `if` and `for` for runtime code, and template metaprogramming for compile-time code), Coq makes use of its type system.

```
 1  Parameter RA : Type -> Type.
 2  Variable  S  : Type.
 3
 4  Parameters
 5    (single  : S -> RA S)
 6    (models  : Ensemble S -> RA S -> Prop)
 7    (add     : forall {I : Type} (R : I -> RA S), RA S)
 8    (mul     : forall {I : Type} (R : I -> RA S), RA S)
 9    (top     : RA S)
10    (bottom  : RA S)
11    (implies : relation (RA S)).
```

Listing C.18: Result Algebra Signature in Coq

C.6 Clarifications

In this section, we take a closer look at some Coq fragments presented in the text.

C.6.1 Result Algebra Definitions

Section 9.3.2 discusses the result algebra. Listing 9.1 (p. 111), repeated here in Listing C.18 for convenience, shows the Coq definitions of the result algebra operations. We discuss their definitions in turn.

- `RA` on line 1 corresponds to $-^\star$. For example, the type of a result involving natural numbers, written mathematically as $\mathbb{N}^\star$, is written `RA nat` in Coq.
- Line 2 locally introduces a type `S`. It can be seen as implicitly introducing an extra agument to all following definitions which refer to `S`. For example, the "true" type of `single` is `forall S : Type, S -> RA S`.
- Line 5 defines `single` which corresponds to $[\![-]\!]$. For example, the result $[\![5]\!]$ is written `single nat 5` in Coq.
- $- \models -$ is represented by `models` in Coq. `Ensemble A` is defined as `A -> Prop` in Coq's standard library, i.e. it is a characteristic function for a set of items with type `A`.

- Lines 7 and 8 define $\oplus$ and $\otimes$, respectively. `R` is a function representing the expression over which the angelic or demonic choice is taken. Definition 9.3.1 makes use of an indexing set, which is seemingly nowhere to be found in the Coq definitions. The indexing set is actually represented by the domain of `R`. For example, consider the result $\bigotimes_{n\in\{1,2,3\}} [\![n]\!]$: it can be rewritten as

$$\bigotimes_{n\in\{1,2,3\}} [\![n]\!] = \bigotimes_{n\in\{1,2,3\}} (\lambda\, k : \mathbb{N}.\ [\![k]\!])\ n = \bigotimes_{n\in\{1,2,3\}} R\ n$$

  where $R\ (k : \mathbb{N}) = [\![k]\!]$. Currently, R's domain is $\mathbb{N}$. Since we only apply it to 1, 2 and 3, we can restrict its domain: let $R' = R|_{\{1,2,3\}}$, then

$$\bigotimes_{n\in\{1,2,3\}} R\ n = \bigotimes_{n\in\{1,2,3\}} R'\ n = \bigotimes_{n\in\mathrm{dom}\ R'} R'\ n$$

  The indexer and indexing variable now have become redundant. We can define a shorthand notation as follows:

$$\bigotimes R \equiv \bigotimes_{x\in\mathrm{dom}\ R} R\ x$$

  This is exactly how $\otimes$ has been implemented in Coq. This approach does not impose any limitations: `R`'s domain type is `Type`, which allows us the describe any domain we wish. For example, we can write $\bigotimes_{n\in\{1,2,3\}} [\![n]\!]$ as

```
mul (fun x : { x | In x [1;2;3] } => RAsingle (proj1_sig x))
```

  where `{ x | P x }` means "an x for which P holds" and `proj1_sig` extracts the `x` from a `{ x | P x }` object, since it is actually a pair of `x` and a proof object showing that `P x` holds.

- Lines 9–10 are straightforward: they define the existence of $\top$ and $\bot$.

- Line 11 defines a binary relation `implies` between results, i.e. $\Rrightarrow$.

C.6.2 Result Algebra Axioms

In this section, we take a closer look at the Coq implementation of the result algebra axioms (Def. 9.3.1) as shown in Listing 9.2, repeated in Listing C.19. Most of the Coq definitions should be easily understandable.

- Lines 1–3 define Ax-Single.

```
 1 Axiom single_axiom :
 2   forall (s : S) (S : Ensemble S),
 3     models S single s <-> In s S.
 4
 5 Axiom top_axiom : forall (R : False -> RA S),
 6   equiv top (mul R).
 7
 8 Axiom bottom_axiom : forall (R : False -> RA S),
 9   equiv bottom (add R).
10
11 Axiom add_axiom :
12   forall I (R : I -> RA S) (S : Ensemble S),
13     models S (add R) <-> exists i, models S (R i).
14
15 Axiom mul_axiom :
16   forall I (R : I -> RA S) (S : Ensemble S),
17     models S (mul R) <-> forall i, models S (R i).
18
19 Axiom implies_axiom : forall (R R' : RA S),
20   implies R R' <-> forall (S : Ensemble S),
21                      models S R -> models S R'.
22
23 Axiom monotonic_models_axiom :
24   forall (R : RA S) (S S' : Ensemble S),
25     Included S' S -> models S' R -> models S R.
```

Listing C.19: Result Algebra Axioms in Coq

- Lines 5–9 correspond to Def. 9.3.7 (p. 114) and Def. 9.3.8 (p. 115) for $\top$ and $\bot$, respectively. Notice how the domain of `R` is `False`: this type is uninhabited (since it cannot be proven) and thus corresponds to an empty domain.
- Lines 11–17 correspond to axioms Ax-Angelic and Ax-Demonic, respectively.
- Lines 19–21 state Def. 9.3.3 as an axiom.
- Lines 23–25 correspond to axiom Ax-Monotonicity.

```
 1  Parameter RA : Set -> Set.
 2
 3  Variables
 4    (S : Set)
 5    (S_eqdec : forall s s' : S, { s = s' } + { s <> s' }).
 6
 7  Parameters (single     : S -> RA S)
 8             (models     : Ensemble S -> RA S -> Prop)
 9             (add mul    : list (RA S) -> RA S)
10             (top bottom : RA S)
11             (implies    : relation (RA S))
12             (is_bottom  : RA S -> bool).
```

Listing C.20: Effective Result Algebra Signature in Coq

C.6.3 Effective Result Algebra

In Sect. 9.6 (p. 130), we discussed the effective result algebra, a computable variant of the result algebra. For convenience, we repeat the Coq definitions in Listing C.20 and Listing C.21. We explain the differences between the Coq definitions for the noneffective (Listing C.18) and effective result algebras (Listing C.20).

- `RA` is defined on line 1 as `Set -> Set` instead of `Type -> Type`. This ensures that results can only operate on computable data structures.
- `S_eqdec` (line 5) expresses the requirement that equality between elements of `S` should be decidable, i.e. that there exists an algorithm that determines (in finite time) whether two elements are equal or not.
- `models` on line 8 still makes use of `Ensemble` (which is not a computable data structure) and `Prop`. This is allowed as the $\models$ relation is not needed for executing the symbolic execution. We do need to be able, though, to determine at runtime whether a result is equivalent with $\perp$ or not. For this, we introduced `is_bottom` (line 12).
- `add` and `mul` (line 9) are defined in terms of `list`s, i.e. finite cycleless linked lists. This corresponds to having a finite indexing set:

$$\texttt{mul [R1;R2;R3]} = \bigotimes_{i \in \mathbb{Z}_3} R_i$$

The axioms differ only slightly between the noneffective (Listing C.19) and effective result algebra (Listing C.21).

- The effective Coq implementation for Ax-Single has remained unchanged.
- The axioms for $\top$ and $\bot$ have to be adapted (lines 7–11) to operate on lists. We used `ListSet` from Coq's standard library, meaning that `empty_set` is actually just the empty list.
- Instead of being given a function $R : A \to B^\star$ and universally or existentially quantifying over A, lines 13–21 rely on `Set_in` to express the equivalent with lists.
- Lines 23–30, containing the Coq implementations of the axioms for $\Rrightarrow$ and monotonicity, are identical to their noneffective counterparts.

C.6.4 Inductive Formulae Model

In Sect. 9.6.1 (p. 133), we discussed a specific result algebra model. Listing 9.5 (repeated in Listing C.22 for convenience). We explain the code in more detail here.

- Lines 1–4 define a formula as an inductive type. `f_and` and `f_or` correspond to $\otimes$ and $\oplus$, respectively, and these links are defined on lines 27–31. `f_lit` corresponds to $[\![-]\!]$, which is expressed on line 18. For example, $\bigotimes_{n\in\mathbb{N}} [\![n]\!]$ can be written as `f_and nat nat (f_lit nat)`.
- Line 6 defines `empty_R`, where `empty_R A` is the (unique) function with empty domain and range `A`.
- Lines 8–12 define `f_true` and `f_false` by taking the demonic and angelic choice, respectively, over an empty index set. There are linked with $\top$ and $\bot$, respectively, on lines 33-35.
- Lines 20–25 define the $\models$ relation as a fixpoint, i.e. a recursive function which is guaranteed to terminate.
- Lines 37–38 define the $\Rrightarrow$ relation.

This definition of formulae allows us to build trees with infinite branching factor (by taking I to be an infinite set) but only of finite depth. However, we claimed that the concrete execution potentially produces trees of infinite depth. How can we reconcile these contradicting facts?

```
Variable S : Set.

Axiom single_axiom :
  forall (s : S) (S : Ensemble S),
    models S (single s) <-> In s S.

Axiom top_axiom :
  equiv top (mul (empty_set (RA S))).

Axiom bottom_axiom :
  equiv bottom (add (empty_set (RA S))).

Axiom add_axiom :
  forall (Rs : list (RA S)) (S : Ensemble S),
    models S (add Rs) <-> exists R, set_In R Rs /\
                                     models S R.

Axiom mul_axiom :
  forall (Rs : list (RA S)) (S : Ensemble S),
    models S (mul Rs) <-> forall R, set_In R Rs ->
                                     models S R.

Axiom implies_axiom :
  forall (R R' : RA S),
    implies R R' <-> forall (S : Ensemble S),
                          models S R -> models S R'.

Axiom monotonic_models_axiom :
  forall (R : RA S) (S S' : Ensemble S),
    Included S' S -> models S' R -> models S R.
```

Listing C.21: Effective Result Algebra Axioms in Coq

```
Inductive formula (S : Type) : Type :=
| f_and : forall I (R : I -> formula S), formula S
| f_or  : forall I (R : I -> formula S), formula S
| f_lit : S -> formula S.

Definition empty_R (S : Type) : False -> formula S.

Definition f_true  (S : Type) : formula S :=
  (@f_and S False (empty_R S)).

Definition f_false (S : Type) : formula S :=
  (@f_or S False (empty_R S)).

Definition RA (S : Type) : Type := formula S.

Variable S : Type.

Definition single (x : S) : RA S := f_lit x.

Fixpoint models (S : Ensemble S) (R : RA S) : Prop :=
  match R with
    | f_and J R' => forall j : J, models S (R' j)
    | f_or  J R' => exists j : J, models S (R' j)
    | f_lit s    => In s S
  end.

Definition add {I : Type} (R : I -> RA S) : RA S :=
  f_or R.

Definition mul {I : Type} (R : I -> RA S) : RA S :=
  f_and R.

Definition top    : RA S := f_true S.

Definition bottom : RA S := f_false S.

Definition implies (R R' : RA S) : Prop :=
  forall S, models S R -> models S R'.
```

Listing C.22: Inductive Formulae Model in Coq

The explanation is simple: the concrete execution actually only yields trees with finite depth, which is a consequence of Def. 9.8.26 (p. 149). It states

$$\mathsf{c\text{-}execute}(c) = \bigotimes_{n \in \mathbb{N}} \mathsf{c\text{-}execute}_n(c)$$

In other words, c-execute returns only “prefix trees” with finite depth of the actual potentially infinitely deep execution tree.

Appendix D

Coq Scripts

This chapter contains the all Coq scripts. While the proofs themselves are all machine checked by Coq, it is still possible we proved the wrong theorems. The reason for the inclusion of the scripts in this thesis is to give the reader a chance to check the right theorems were proven.

The proofs have been left out for multiple reasons:

- As mentioned above, they are machine checked.
- All proofs are written in Coq's tactic language, which has not been designed with readability in mind.
- The most interesting proofs are included in the main body of this thesis.
- Full Coq scripts (i.e. with proofs) are available online [110].

D.1 Assertion

```
Require Import Notations.
Require Import ListExt.
Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Predicate.

Import BooleanExpression.Notations.

Inductive t : Set :=
| bexpr  : BooleanExpression.t → t
| sepand : t → t → t
| cond   : BooleanExpression.t → t → t → t
| pred   : Predicate.t → Expression.t → Id.t → t.

Definition true : t := bexpr BooleanExpression.true.

Inductive assertion_predicate_reference : t → Predicate.t → Prop :=
| apr_pred : forall p e x,
             assertion_predicate_reference (pred p e x) p
| apr_sepand_left  : forall p a1 a2,
                     assertion_predicate_reference a1 p →
                     assertion_predicate_reference (sepand a1 a2) p
| apr_sepand_right : forall p a1 a2,
                     assertion_predicate_reference a2 p →
                     assertion_predicate_reference (sepand a1 a2) p
| apr_cond_then : forall p b a1 a2,
                  assertion_predicate_reference a1 p →
                  assertion_predicate_reference (cond b a1 a2) p
| apr_cond_else : forall p b a1 a2,
                  assertion_predicate_reference a2 p →
                  assertion_predicate_reference (cond b a1 a2) p.

Module Notations.
  Delimit Scope assertion_scope with assertion.
  Bind Scope assertion_scope with t.

  Coercion bexpr : BooleanExpression.t >-> t.

  Notation "a ★ a'" :=
    (sepand a a') (at level 75) : assertion_scope.

  Notation "'If' b 'Then' a1 'Else' a2" :=
    (cond b a1 a2) (at level 80) : assertion_scope.

  Notation "p [ e ; ? y ]" :=
    (pred p e y) (at level 50, e at level 0, y ident) : assertion_scope.

  Notation "p ↦ ? q" :=
    (pred Predicate.ptr p q) (at level 50, q ident) : assertion_scope.

  Arguments Scope cond [ bexpr_scope assertion_scope assertion_scope ].
  Arguments Scope sepand [ assertion_scope assertion_scope ].
  Open Scope assertion_scope.
End Notations.
```

D.2 AssocList

```
Require Import Notations.
Require Import List.

Set Implicit Arguments.

Module Type EQ_DEC.
  Parameters
    (t : Set)
    (eq_dec : forall x y : t, { x = y } + { x ≠ y }).
End EQ_DEC.

Module Make (M : EQ_DEC).
  Definition key := M.t.

  Section Value.
    Variable value : Set.

    Definition t : Set := list (key * value).

    Definition empty := @nil (key * value).

    Definition add (xs : t) (x : key) (y : value) : t := (x, y) :: xs.

    Fixpoint lookup (xs : t) (k : key) (v : value) : value :=
      match xs with
        | (x, y)::xs => if M.eq_dec x k then y else lookup xs k v
        | nil        => v
      end.

    Fixpoint lookup_option (xs : t) (k : key) : option value :=
      match xs with
        | (x, y)::xs => if M.eq_dec x k then Some y else lookup_option xs k
        | nil        => None
      end.

    Definition keys (xs : t) : list key :=
      map (@fst _ _) xs.

    Definition values (xs : t) : list value :=
      map (@snd _ _) xs.

    Theorem lookup_add :
      forall xs k v v', lookup (add xs k v) k v' = v.
    Proof. (* 4 lines *) Qed.

    Theorem lookup_other :
      forall xs k k' v v', k ≠ k' → lookup (add xs k v) k' v' = lookup xs k' v'.
    Proof. (* 4 lines *) Qed.

    Theorem lookup_default :
      forall xs k v,
       forallb (fun p => if M.eq_dec (fst p) k then false else true) xs = true →
       lookup xs k v = v.
    Proof. (* 11 lines *) Qed.
  End Value.
End Make.
```

D.3 BooleanExpression

```
Require Import Notations.
Require Import Arith.
Require Import Sumbool.
Require Import Identifier.
Require Expression.

Inductive t : Set :=
| eq   : Expression.t  → Expression.t  → t
| lt   : Expression.t  → Expression.t  → t
| le   : Expression.t  → Expression.t  → t
| and  : t → t  → t
| not  : t → t
| true : t.

Fixpoint evaluate (b : t) (s : Id.t → nat) : bool :=
  let aux : Expression.t → nat :=
    fun e : Expression.t => Expression.evaluate e s in
      match b with
        | eq e e'  => proj1_sig (bool_of_sumbool (eq_nat_dec (aux e)
                                                             (aux e')))
        | lt e e'  => proj1_sig (bool_of_sumbool (lt_dec (aux e)
                                                         (aux e')))
        | le e e'  => proj1_sig (bool_of_sumbool (le_dec (aux e)
                                                         (aux e')))
        | and b b' => andb (evaluate b s) (evaluate b' s)
        | not b    => negb (evaluate b s)
        | true     => Datatypes.true
      end.

Definition eq_dec : forall b b' : t, {b = b'} + {b ≠ b'} .
  induction b as [ e e0 | e e0 | e e0 | b rec b0 rec' | b | ];
    destruct b' as [ e1 e2 | e1 e2 | e1 e2 | b1 b2 | b' | ];
      try (right; discriminate; fail).

  (* eq *)
  destruct (Expression.eq_dec e e1); destruct (Expression.eq_dec e0 e2); subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.

  (* lt *)
  destruct (Expression.eq_dec e e1); destruct (Expression.eq_dec e0 e2); subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.

  (* le *)
  destruct (Expression.eq_dec e e1); destruct (Expression.eq_dec e0 e2); subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.

  (* and *)
  destruct (rec b1) as [ eql | neql ];
    destruct (rec' b2) as [ eql' | neql' ]; subst;
      try (left; reflexivity; fail);
        right; intro X; try (elim neql);
          try (elim neql'); injection X; trivial.
  (* not *)
  destruct (IHb b');
    [ left; subst; trivial | right; intro; elim n; injection H; trivial ].

  (* true *)
  left; trivial.
Defined.

Definition beq (b b' : t) : bool.
  destruct (eq_dec b b').
  exact Datatypes.true.
  exact false.
Defined.

Module Notations.

  Delimit Scope bexpr_scope with bexpr.
  Bind Scope bexpr_scope with t.

  Infix "==" := eq (at level 70) : bexpr_scope.
  Infix "<"  := lt : bexpr_scope.
  Notation "¬ e" := (not e) : bexpr_scope.

  Open Scope bexpr_scope.

End Notations.
```

D.4 Chunk

```
Require Import Notations.

Require Import EqDec.
Require Import List.
Require Term.
Require Predicate.

Module Make (Arg : EQDEC).

  Inductive t' : Set :=
  | Chunk : Predicate.t → Arg.t → Arg.t → t'.

  Definition t := t'.

  Theorem eq_dec (c c' : t) : { c = c' } + { c ≠ c' } .
  Proof. (* 10 lines *) Qed.

  Definition beq (c c' : t) : bool :=
    if eq_dec c c' then true else false.

  Definition args (c : t) : list Arg.t :=
    match c with
      | Chunk _ x y => x :: y :: nil
    end.

  Module Notations.

    Delimit Scope chunk_scope with chunk.
    Bind Scope chunk_scope with t.

    Notation "'mb' [ t ; n ]" := (Chunk Predicate.mb t n)
      (t at level 0, n at level 0) : chunk_scope.

    Notation "x ↦ y" :=
      (Chunk Predicate.ptr x y) (at level 40) : chunk_scope.

  End Notations.

End Make.
```

D.5 ConcreteExecution

```
Require Import Notations.
Require Import Basics.
Require Import EnsembleExt.
Require Import Arith.
Require Import String.
Require Import ListExt.
Require Import ListSet.
Require Import Sumbool.
Require RADefinitions.
Require RAAxioms.
Require RAOperators.

Require Nat.
Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Routine.
Require Store.
Require Heap.
Require Predicate.
Require Chunk.
Require SIL.

Open Scope program_scope.
Open Scope bool_scope.

Module Make
  (Import RAD : RADefinitions.DEFINITIONS)
  (Import RAA : RAAxioms.AXIOMS RAD)
  (Import RAO : RAOperators.AXIOMS RAD RAA).

  Module RAN := RANotations.Make(RAD).
  Include RAN.

  Module RAOPS := RAOperators.Make RAD RAA RAO.

  Import RAOPS.
  Import RAOPS.DoNotation.
  Import RAOPS.Util.

  Module RName  := Routine.DefaultName.
  Module CStore := Store.AssocListStore Nat.
  Module CChunk := Chunk.Make Nat.
  Module CHeap  := Heap.Default CChunk.

  Module StoreNotations := Store.Notations Nat CStore.

  Import BooleanExpression.Notations.
  Import StoreNotations.
  Import CChunk.Notations.
  Import SIL.Notations.

  Definition zero_store := CStore.constant_store 0.
  Notation "'s_0'" := zero_store : store_scope.
```

```
(*
   Concrete state
*)

Inductive concrete_state : Set :=
  ConcreteState : CStore.t → CHeap.t → concrete_state.

Notation "⟨ s , h ⟩" := (ConcreteState s h) (at level 0).

Definition store : field concrete_state CStore.t :=
  (fun σ => match σ with
              | ⟨s, _⟩ => s
            end,
   fun σ s => match σ with
                | ⟨_, h⟩ => ⟨s, h⟩
              end).

Definition heap : field concrete_state CHeap.t :=
  (fun σ => match σ with
              | ⟨_, h⟩ => h
            end,
   fun σ h => match σ with
                | ⟨s, _⟩ => ⟨s, h⟩
              end).

Definition state : field concrete_state concrete_state :=
  ((fun σ => σ), (fun σ σ' => σ')).

Open Scope op_scope.

Definition c_operator := operator concrete_state concrete_state.

Definition read_store (id : Id.t) : c_operator nat :=
  with_current store (CStore.lookup id).

Definition update_store (id : Id.t) (n : nat) : c_operator unit :=
  update_current store (CStore.bind id n).

Definition evaluate (e : Expression.t) : c_operator nat :=
  with_current store (Expression.evaluate e ∘ flip CStore.lookup).

Definition with_store
  {A : Set} (s : CStore.t) (op : c_operator A) : c_operator A :=
  s' ← current store;
  _  ← set_current store s;
  r  ← op;
  _  ← set_current store s';
  yield r.

Definition assume_bexpr (b : BooleanExpression.t) : c_operator unit :=
  r ← with_current store (BooleanExpression.evaluate b ∘ flip CStore.lookup);
  assume r.

Definition assert_bexpr (b : BooleanExpression.t) : c_operator unit :=
  r ← with_current store (BooleanExpression.evaluate b ∘ flip CStore.lookup);
  assert r.

Definition produce_chunk (α : CChunk.t) : c_operator unit :=
  update_current heap (CHeap.produce α).
```

```
Definition consume_chunk (α : CChunk.t) : c_operator unit :=
  r ← with_current heap (CHeap.consume α);
  match r with
    | Some h’ => set_current heap h’
    | None    => failure
  end.

Definition pick_chunk_angelically : c_operator CChunk.t :=
  αs ← with_current heap CHeap.enum;
  α ← pick_angelically (FromList αs);
  yield (proj1_sig α).

Definition find_chunk (pred : Predicate.t) (ℓ : nat) : c_operator CChunk.t :=
  α ← pick_chunk_angelically;
  match α with
    | CChunk.Chunk p x y => assert (Predicate.beq p pred);
                            assert (beq_nat x ℓ);
                            yield (CChunk.Chunk p x y)
  end.

Definition read_cell (ℓ : nat) : c_operator nat :=
  α ← find_chunk Predicate.ptr ℓ;
  match α with
    | CChunk.Chunk _ _ v => yield v
  end.

Definition write_cell (ℓ v : nat) : c_operator unit :=
  α ← find_chunk Predicate.ptr ℓ;
  consume_chunk α;
  produce_chunk (ℓ ↦ v).

Definition clear_heap : c_operator unit :=
  set_current heap CHeap.empty.

Definition leak_check : c_operator unit :=
  r ← with_current heap CHeap.is_empty; assert r.

Definition alloc_set : c_operator (list nat) :=
  let get_loc :=
    fun α => match α with
               | CChunk.Chunk p ℓ _ =>
                 if Predicate.beq p Predicate.ptr
                   then ℓ :: nil
                   else nil
             end
    in
    h ← current heap;
    yield (concat_map get_loc h).

Fixpoint pick_demonically_n (n : nat) (A : Type) : c_operator (list A) :=
  match n with
    | O   => yield nil
    | S n => v ← pick_demonically A;
             vs ← pick_demonically_n n A;
             yield (v :: vs)
  end.
```

```
Definition allocate (n : nat) : c_operator nat :=
  let allocate_at :=
    fix aux (ℓ : nat) (vs : list nat) : c_operator unit :=
    match vs with
      | nil   => nop
      | v::vs =>
        let k := length vs in
        A ← alloc_set;
        assume (proj1_sig (bool_of_sumbool (in_dec eq_nat_dec (ℓ + k) A)));
        produce_chunk ((ℓ + k) ↦ v)%chunk;
        aux ℓ vs
    end
  in
  ns ← pick_demonically_n (S n) nat;
  match ns with
    | ℓ::vs => allocate_at ℓ vs;
               produce_chunk (mb [ℓ; n])%chunk;
               yield ℓ
    | nil   => failure
  end.

Fixpoint consume_cells (ℓ : nat) (n : nat) : c_operator unit :=
  match n with
    | O   => nop
    | S n => α ← find_chunk Predicate.ptr (ℓ + n);
             consume_chunk α;
             consume_cells ℓ n
  end.

Definition block_size (ℓ : nat) : c_operator nat :=
  α ← find_chunk Predicate.mb ℓ;
  match α with
    | CChunk.Chunk _ _ n => yield n
  end.

Section WithProgram.
  Variable (Π : SIL.program).

  Hypothesis (wellformed : SIL.wellformed_program Π).

  Open Scope command_scope.

  Fixpoint concrete_execution_n
    (c : SIL.command) (n : nat) : c_operator unit :=
    match n with
      | O => block
      | S n =>
        match c with
          | skip => yield □
          | x := e =>
             v ← evaluate e;
             update_store x v
          | c; c' =>
             concrete_execution_n c n;
             concrete_execution_n c' n
          | x := malloc n =>
```

```
                ℓ ← allocate n;
                update_store x ℓ
            | free e => ℓ ← evaluate e;
                        n ← block_size ℓ;
                        consume_cells ℓ n;
                        consume_chunk (mb [ℓ; n])
            | ⟦e⟧ := e' =>
                ℓ ← evaluate e;
                v ← evaluate e';
                write_cell ℓ v
            | x := ⟦e⟧ =>
                ℓ ← evaluate e;
                v ← read_cell ℓ;
                update_store x v
            | If b Then c Else c' =>
                let then_clause :=
                  assume_bexpr b;
                  concrete_execution_n c n in
                let else_clause :=
                  assume_bexpr (¬ b)%bexpr;
                  concrete_execution_n c' n in
                let clauses := then_clause :: else_clause :: nil in
                op ← pick_demonically (FromList clauses);
                proj1_sig op
            | r[e] => v ← evaluate e;
                      rdef ← from_some (SIL.routines Π r);
                      let x := SIL.argument rdef in
                      let body := SIL.routine_body rdef in
                      with_store (s_0⟦x := v⟧)%store
                                 (concrete_execution_n body n)
          end
      end.

    Definition concrete_execution (c : SIL.command) : c_operator unit :=
      n ← pick_demonically nat;
      concrete_execution_n c n.

    Close Scope command_scope.

    Definition verify : c_operator unit :=
      v ← pick_demonically nat;
      r ← from_some (SIL.routines Π (SIL.main_routine Π));
      update_store (SIL.argument r) v;
      concrete_execution (SIL.routine_body r).

    Definition valid_program : Prop :=
      not (verify ⟨s_0, CHeap.empty⟩ ⟺ ⊥).
  End WithProgram.
  Close Scope op_scope.
End Make.
```

D.6 DependentProduct

```
Require Import EnsembleExt.
Require Import Notations.
Require Export Classical.
Require Import Decidable.
Require Export Logic.ClassicalChoice.
Require Import Misc.

Inductive dependent_product {A B} (R : A → Ensemble B) : Ensemble (A → B) :=
  In_dependent_product : ∀ f : A → B,
    (∀ i, f i ∈ R i) → f ∈ dependent_product R.

Notation "'Π' i, R" :=
  (dependent_product (fun i => R)) (at level 65, i ident).

Notation "'Π' i : A , R" :=
  (@dependent_product A _ (fun i => R)) (at level 65, i ident).

Theorem dependent_product_empty_A :
  forall A B (R : A → Ensemble B) (H : ¬ inhabited A),
    Same_set _ (dependent_product R) (Singleton (function_with_empty_domain A B
H)).
Proof. (* 20 lines *) Qed.

Theorem dependent_product_empty_result :
  forall A B (R : A → Ensemble B),
    Same_set _ (dependent_product R) ∅ ↔ exists a, Same_set _ (R a) ∅.
Proof. (* 38 lines *) Qed.

Theorem dependent_product_Included :
  forall A B (R R' : A → Ensemble B),
    (∀ a, R a ⊆ R' a) → dependent_product R ⊆ dependent_product R'.
Proof. (* 7 lines *) Qed.

Theorem dependent_product_singleton_Rx :
  forall A B (f : A -> B),
    Same_set _ (dependent_product (fun (x : A) => Singleton (f x)))
               (Singleton f).
Proof. (* 22 lines *) Qed.
```

D.7 EInductiveFormulae

```
Require Import Notations.
Require Import EnsembleExt.
Require Import Misc.
Require Import ListExt.
Require Import Permutation.
Require Import Relations.
Require Import ListSetExt.
Require ERADefinitions.
Require ERANotations.
Require ERAAxioms.
Require ERAOperators.
Require Import ChoiceFacts.

Set Implicit Arguments.

Module MakeDefinitions <: ERADefinitions.DEFINITIONS.

  Inductive formula {Σ : Set} : Set :=
  | f_and    : list (@formula Σ) → @formula Σ
  | f_or     : list (@formula Σ) → @formula Σ
  | f_single :                Σ  → @formula Σ.

  Section InductionHypothesis.
    Variable Σ : Set.

    Variable P : @formula Σ → Prop.

    Hypothesis H_single : forall σ : Σ, P (f_single σ).

    Hypothesis H_or : forall xs : list formula, Forall P xs → P (f_or xs).

    Hypothesis H_and : forall xs : list formula, Forall P xs → P (f_and xs).

    Fixpoint formula_ind' (f : formula) : P f :=
      match f with
        | f_single x => H_single x
        | f_or xs =>
          @H_or xs ((fix aux (xs : list formula) : Forall P xs :=
                      match xs with
                        | nil =>
                          @Forall_nil _ P
                        | x::xs =>
                          @Forall_cons _ P x xs (formula_ind' x) (aux xs)
                      end) xs)
        | f_and xs =>
          @H_and xs ((fix aux (xs : list formula) : Forall P xs :=
                       match xs with
                         | nil =>
                           @Forall_nil _ P
                         | x::xs =>
                           @Forall_cons _ P x xs (formula_ind' x) (aux xs)
                       end) xs)
      end.
  End InductionHypothesis.
```

```
Section Recursive.
  Variable Σ : Set.

  Variable R : Type.

  Variable F : @formula Σ → R.

  Variable F_single : Σ → R.

  Hypothesis F_or : list R → R.

  Hypothesis F_and : list R → R.

  Fixpoint formula_rec' (f : formula) : R :=
    match f with
      | f_single x => F_single x
      | f_or xs => F_or (map formula_rec' xs)
      | f_and xs => F_and (map formula_rec' xs)
    end.
End Recursive.

Definition sum (ns : list nat) : nat :=
  @fold_left nat nat plus ns 0.

Definition formula_metric {Σ : Set} (f : @formula Σ) : nat :=
  @formula_rec' Σ nat (fun _ => 1) sum sum f.

Definition 𝓡 := @formula.

Section StateSection.
  Variables
    (Σ : Set)
    (Σ_eqdec : forall σ σ' : Σ, { σ = σ' } + { σ ≠ σ' }).

  Definition single := @f_single Σ.

  Definition add := @f_or Σ.

  Definition mul := @f_and Σ.

  Definition top := @f_and Σ (empty_set (𝓡 Σ)).

  Definition bottom := @f_or Σ (empty_set (𝓡 Σ)).

  Fixpoint is_bottom (R : 𝓡 Σ) : bool :=
    match R with
      | f_single σ => false
      | f_or Rs =>
        (fix rec (Rs : list (𝓡 Σ)) :=
          match Rs with
            | R :: Rs => if is_bottom R then rec Rs else false
            | nil     => true
          end) Rs
      | f_and Rs =>
        (fix rec (Rs : list (𝓡 Σ)) :=
          match Rs with
            | R :: Rs => if is_bottom R then true else rec Rs
            | nil     => false
          end) Rs
    end.
    Definition models (S : Ensemble Σ) (R : 𝓡 Σ) : Prop :=
      @formula_rec' Σ Prop (fun σ => σ ∈ S)
                           (Exists id)
                           (Forall id) R.

    Definition implies (R R' : 𝓡 Σ) : Prop :=
      forall S, models S R → models S R'.

    Definition equiv : relation (𝓡 Σ) :=
      fun R R' => implies R R' ∧ implies R' R.
  End StateSection.
End MakeDefinitions.

Module MakeAxioms <: ERAAxioms.AXIOMS (MakeDefinitions).

  Include MakeDefinitions.

  Module RAN := ERANotations.Make( MakeDefinitions ).

  Include RAN.

  Section StateSection.

    Variable Σ : Set.

    Theorem single_axiom :
      forall (s : Σ) (S : Ensemble Σ), S ⊨ single s ↔ s ∈ S.
    Proof. (* 3 lines *) Qed.

    Theorem top_axiom : ⊤ ⟺ ⊗ (empty_set (𝓡 Σ)).
    Proof. (* 1 lines *) Qed.

    Theorem bottom_axiom :
      ⊥ ⟺ ⊕ (empty_set (𝓡 Σ)).
    Proof. (* 1 lines *) Qed.

    Lemma add_models_hd :
      forall (R : 𝓡 Σ) (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ R → S ⊨ ⊕(R::Rs).
    Proof. (* 4 lines *) Qed.

    Lemma add_models_tl :
      forall (R : 𝓡 Σ) (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ ⊕ Rs → S ⊨ ⊕(R::Rs).
    Proof. (* 4 lines *) Qed.

    Theorem add_axiom :
      forall (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ ⊕ Rs ↔ ∃ R, set_In R Rs ∧ S ⊨ R.
    Proof. (* 39 lines *) Qed.

    Lemma mul_models_cons :
      forall (R : 𝓡 Σ) (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ R ∧ S ⊨ ⊗ Rs ↔ S ⊨ ⊗ (R::Rs).
    Proof. (* 13 lines *) Qed.

    Theorem mul_axiom :
      forall (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ ⊗ Rs ↔ ∀ R, set_In R Rs → S ⊨ R.
    Proof. (* 29 lines *) Qed.
    Theorem implies_axiom :
      forall (R R' : 𝓡 Σ),
        R ⇛ R' ↔ forall (S : Ensemble Σ), S ⊨ R → S ⊨ R'.
    Proof. (* 2 lines *) Qed.

    Theorem monotonic_models_axiom :
      forall (R : 𝓡 Σ) (S S' : Ensemble Σ),
        S' ⊆ S → S' ⊨ R → S ⊨ R.
    Proof. (* 27 lines *) Qed.

  End StateSection.

End MakeAxioms.

Module MakeOperatorAxioms
  <: ERAOperators.AXIOMS(MakeDefinitions)(MakeAxioms).

  Import MakeDefinitions.
  Import MakeAxioms.

  Module RAN := ERANotations.Make(MakeDefinitions).
  Include RAN.

  Section StateSection.

    Variable Σ Σ' : Set.

    Fixpoint lift (f : Σ → 𝓡 Σ') (R : 𝓡 Σ) : 𝓡 Σ' :=
      match R with
        | f_single x => f x
        | f_and Rs => f_and (map (lift f) Rs)
        | f_or Rs => f_or (map (lift f) Rs)
      end.

    Theorem lift_models :
      forall R S S' (f : Σ → 𝓡 Σ'),
        S ⊨ R → (forall σ, σ ∈ S → S' ⊨ f σ) →
        S' ⊨ lift f R.
    Proof. (* 67 lines *) Qed.

    Theorem lift_models' :
      forall R S' (f : Σ → 𝓡 Σ'),
        S' ⊨ lift f R →
        exists S, S ⊨ R ∧ (forall σ, σ ∈ S → S' ⊨ f σ).
    Proof. (* 111 lines *) Qed.

    Theorem monotonic_lift_axiom :
      forall (f g : Σ → 𝓡 Σ') (R R' : 𝓡 Σ),
        (forall Σ, f Σ ⇛ g Σ) → R ⇛ R' → lift f R ⇛ lift g R'.
    Proof. (* 16 lines *) Qed.

    Theorem implies_reflexivity :
      forall R : 𝓡 Σ, R ⇛ R.
    Proof. (* 4 lines *) Qed.

    Theorem equiv_reflexivity :
      forall R : 𝓡 Σ, R ⟺ R.
    Proof. (* 3 lines *) Qed.
    Theorem lift_add_axiom :
      forall (f : Σ → ℛ Σ’) (Rs : list (ℛ Σ)),
        lift f (add Rs) ⟺ add (List.map (lift f) Rs).
    Proof. (* 12 lines *) Qed.

    Theorem lift_mul_axiom :
      forall (f : Σ → ℛ Σ’) (Rs : list (ℛ Σ)),
        lift f (mul Rs) ⟺ mul (List.map (lift f) Rs).
    Proof. (* 12 lines *) Qed.

    Theorem lift_single_axiom :
      forall (f : Σ → ℛ Σ’) (s : Σ),
        lift f (single s) ⟺ f s.
    Proof. (* 11 lines *) Qed.

  End StateSection.

End MakeOperatorAxioms.
```

D.8 ERAAxioms

```
Require Import Notations.
Require Import Ensembles.
Require Import ListExt.
Require Import Permutation.
Require Import Relations.
Require Import ListSetExt.
Require ERADefinitions.
Require ERANotations.

Set Implicit Arguments.

Module Type AXIOMS
  (Import Definitions : ERADefinitions.DEFINITIONS).

  Module RAN := ERANotations.Make ( Definitions ).
  Include RAN.

  Section StateSection.

    Variable Σ : Set.

    Axiom single_axiom :
      forall (s : Σ) (S : Ensemble Σ),
        S ⊨ single s ↔ s ∈ S.

    Axiom top_axiom :
      ⊤ ⟺ ⊗ (empty_set (𝓡 Σ)).

    Axiom bottom_axiom :
      ⊥ ⟺ ⊕ (empty_set (𝓡 Σ)).

    Axiom add_axiom :
      forall (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ ⊕ Rs ↔ ∃ R, set_In R Rs ∧ S ⊨ R.

    Axiom mul_axiom :
      forall (Rs : list (𝓡 Σ)) (S : Ensemble Σ),
        S ⊨ ⊗ Rs ↔ ∀ R, set_In R Rs → S ⊨ R.

    Axiom implies_axiom :
      forall (R R' : 𝓡 Σ),
        R ⇛ R' ↔ forall (S : Ensemble Σ), S ⊨ R → S ⊨ R'.

    Axiom monotonic_models_axiom :
      forall (R : 𝓡 Σ) (S S' : Ensemble Σ), S' ⊆ S → S' ⊨ R → S ⊨ R.

  End StateSection.

  Ltac ra_axiom :=
    match goal with
      | |- models _ (add _)     => apply add_axiom
      | |- models _ (mul _)     => apply mul_axiom
      | |- models _ (single _)  => apply single_axiom
      | |- implies _ _          => apply implies_axiom
    end.
  Ltac ra_axiom_in H :=
    match goal with
      | [ H' : models _ (add _) |- _ ] => match H' with
                                            | H => rewrite add_axiom in H
                                          end
      | [ H' : models _ (mul _) |- _ ] => match H' with
                                            | H => rewrite mul_axiom in H
                                          end
      | [ H' : models _ (single _) |- _ ] => match H' with
                                               | H => rewrite single_axiom in H
                                             end
      | [ H' : implies _ _ |- _ ] => match H' with
                                       | H => rewrite implies_axiom in H
                                     end
    end.

  Ltac ra_axioms :=
    match goal with
      | [ H : _ |- _ ] => ra_axiom_in H; ra_axioms
      | _ => (ra_axiom; ra_axioms) || idtac
    end.

End AXIOMS.
```

D.9 ERADefinitions

```
Require Import Notations.
Require Import Ensembles.
Require Import List.
Require Import Relations.
Require Import ListSet.

Set Implicit Arguments.

Module Type DEFINITIONS.

  Parameter ℛ : Set → Set.

  Section StateSection.

    Variables
      (Σ : Set)
      (Σ_eqdec : forall σ σ' : Σ, { σ = σ' } + { σ ≠ σ' }).

    Parameters
      (single     : Σ → ℛ Σ)
      (models     : Ensemble Σ → ℛ Σ → Prop)
      (add mul    : list (ℛ Σ) → ℛ Σ)
      (top bottom : ℛ Σ)
      (implies    : relation (ℛ Σ))
      (is_bottom  : ℛ Σ → bool).

    Definition equiv : relation (ℛ Σ) :=
      fun R R' => implies R R' ∧ implies R' R.

  End StateSection.

  Implicit Arguments top [ Σ ].
  Implicit Arguments bottom [ Σ ].

End DEFINITIONS.
```

D.10 ERANotations

```
Require Import Notations.
Require Import Ensembles.
Require Import ListExt.
Require Import Permutation.
Require Import Relations.
Require Import ListSetExt.
Require ERADefinitions.

Set Implicit Arguments.

Module Make (Import D : ERADefinitions.DEFINITIONS).

  Notation "⊕ Rs" := (@add _ Rs) (at level 0).

  Notation "⊗ Rs" := (@mul _ Rs) (at level 0).

  Notation "⊤" := (@top _).

  Notation "⊥" := (@bottom _).

  Notation "R ⇛ R'" := (implies R R') (at level 70).

  Notation "S ⊨ R" := (models S R) (at level 75).

  Notation "R ⟺ R'" := (equiv R R') (at level 70).

End Make.
```

D.11 ERAOperators

```
Require Import Notations.
Require Import Ensembles.
Require Import List.
Require Import Basics.
Require ERADefinitions.
Require ERANotations.
Require ERAAxioms.

Set Implicit Arguments.

Module Type AXIOMS
  (Import Definitions : ERADefinitions.DEFINITIONS)
  (Import Axioms      : ERAAxioms.AXIOMS Definitions).

  Module RAN := ERANotations.Make(Definitions).
  Include RAN.

  Section StateSection.
    Variable Σ Σ' : Set.

    Parameter lift : (Σ → 𝓡 Σ') → 𝓡 Σ → 𝓡 Σ'.

    Axiom monotonic_lift_axiom :
      forall (f g : Σ → 𝓡 Σ') (R R' : 𝓡 Σ),
        (forall Σ, f Σ ⭆ g Σ) → R ⭆ R' → lift f R ⭆ lift g R'.

    Axiom lift_add_axiom :
      forall (f : Σ → 𝓡 Σ') (Rs : list (𝓡 Σ)),
        lift f (add Rs) ⟺ add (List.map (lift f) Rs).

    Axiom lift_mul_axiom :
      forall (f : Σ → 𝓡 Σ') (Rs : list (𝓡 Σ)),
        lift f (mul Rs) ⟺ mul (List.map (lift f) Rs).

    Axiom lift_single_axiom :
      forall (f : Σ → 𝓡 Σ') (s : Σ),
        lift f (single s) ⟺ f s.
  End StateSection.
End AXIOMS.

Module Make
  (Import Definitions : ERADefinitions.DEFINITIONS)
  (Import Axioms      : ERAAxioms.AXIOMS Definitions)
  (Import Operators   : AXIOMS Definitions Axioms).

  Module RAN := ERANotations.Make(Definitions).
  Include RAN.

  Definition primitive_operator {Σ Σ' : Set} := Σ → 𝓡 Σ'.

  Definition primitive_bind {Σ Σ' Σ" : Set}
    (op  : @primitive_operator Σ Σ')
    (op' : @primitive_operator Σ' Σ") : @primitive_operator Σ Σ" :=
    compose (lift op') op.

  Notation "op ⋙ op'" := (primitive_bind op op') (at level 40).
```

```
Definition operator {Σ Σ' A : Set} :=
  @primitive_operator Σ (A * Σ').

Definition bind {Σ Σ' Σ" A B : Set}
                (op          : @operator Σ Σ' A)
                (f           : A → @operator Σ' Σ" B) :=
  primitive_bind op (prod_curry f).

Definition failure {Σ Σ' : Set} : @primitive_operator Σ Σ' :=
  fun σ : Σ => @bottom Σ'.

Definition block {Σ Σ' : Set} : @primitive_operator Σ Σ' :=
  fun σ : Σ => @top Σ'.

Definition pick_angelically {Σ A : Set} (xs : list A) : @operator Σ Σ A :=
  fun σ : Σ => add (map (compose (@single (A * Σ)) (flip (@pair A Σ) σ)) xs).

Definition pick_demonically {Σ A : Set} (xs : list A) : @operator Σ Σ A :=
  fun σ : Σ => mul (map (compose (@single (A * Σ)) (flip (@pair A Σ) σ)) xs).

Definition nop {Σ : Set} : @operator Σ Σ unit :=
  fun σ : Σ => single (□, σ).

Definition assert {Σ : Set} (b : bool) : @operator Σ Σ unit :=
  match b with
    | true  => nop
    | false => failure
  end.

Definition assume {Σ : Set} (b : bool) : @operator Σ Σ unit :=
  match b with
    | true  => nop
    | false => block
  end.

Definition yield {Σ A : Set} (x : A) : @operator Σ Σ A :=
  fun σ : Σ => single (x, σ).

Definition current_state {Σ : Set} : @operator Σ Σ Σ :=
  fun σ : Σ => single (σ, σ).

Definition set_current_state {Σ : Set} (σ' : Σ) : @operator Σ Σ unit :=
  fun σ : Σ => single (□, σ').

Module DoNotation.
  Delimit Scope op_scope with op.
  Bind Scope op_scope with operator.

  Notation "x ← op ; rest" :=
    (bind op (fun x => rest))
    (at level 100,
      op at level 99,
      rest at level 100,
      right associativity) : op_scope.

  Notation "op ; rest" :=
    (bind op (fun (_ : unit) => rest))
    (at level 100) : op_scope.
End DoNotation.
```

```
  Module Util.

    Open Scope program_scope.

    Import DoNotation.

    Definition field A B := ((A → B) * (A → B → A))%type.

    Definition get {A B} (f : field A B) := fst f.

    Definition set {A B} (f : field A B) := snd f.

    Definition with_ {A B C} (f : field A B) (g : B → C) := g ∘ get f.

    Definition update {A B} (f : field A B) (g : B → B) :=
      fun (s : A) => set f s (g (get f s)).

    Section StateSection.

      Variable (Σ : Set).

      Open Scope op_scope.

      Definition current {Σ A : Set}
                         (f   : field Σ A) : @operator Σ Σ A :=
        σ ← current_state;
        yield (get f σ).

      Definition set_current {A : Set}
                             (f : field Σ A)
                             (x : A) : @operator Σ Σ unit :=
        σ ← current_state;
        set_current_state (set f σ x).

      Definition with_current {A B : Set}
                              (f   : field Σ A)
                              (g   : A → B) : @operator Σ Σ B :=
        x ← current f;
        yield (g x).

      Definition update_current {A : Set}
                                (f : field Σ A)
                                (g : A → A) : @operator Σ Σ unit :=
        x ← current f;
        set_current f (g x).

      Definition from_some {A : Set}
                           (x : option A) : @operator Σ Σ A :=
        match x with
          | Some x => yield x
          | None   => failure
        end.

      Close Scope op_scope.
    End StateSection.
  End Util.
End Make.
```

D.12 ESymbolicExecution

```
Require Import Notations.
Require Import Basics.
Require Import ListExt.
Require Import ListSet.
Require Import Arith.
Require Import String.
Require Import Bool.
Require AssocList.
Require ERADefinitions.
Require ERAAxioms.
Require ERAOperators.

Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Symbol.
Require Routine.
Require Term.
Require Formula.
Require SMT.
Require Store.
Require Heap.
Require Predicate.
Require Chunk.
Require SILPP.
Require Assertion.

Open Scope program_scope.
Open Scope bool_scope.

Module Make
  (Import RAD : ERADefinitions.DEFINITIONS)
  (Import RAA : ERAAxioms.AXIOMS RAD)
  (Import RAO : ERAOperators.AXIOMS RAD RAA).

  Module RAN := ERANotations.Make(RAD).
  Include RAN.

  Module RAOPS := ERAOperators.Make RAD RAA RAO.

  Import RAOPS.
  Import RAOPS.DoNotation.
  Import RAOPS.Util.

  Module Symb   := Symbol.Default.
  Module RName  := Routine.DefaultName.
  Module SStore := Store.AssocListStore Term.
  Module SChunk := Chunk.Make Term.
  Module SHeap  := Heap.Default SChunk.

  Module StoreNotations := Store.Notations Term SStore.

  Import BooleanExpression.Notations.
  Import Term.Notations.
  Import Formula.Notations.
  Import StoreNotations.
  Import SChunk.Notations.
  Import SILPP.Notations.

  Definition zero_store := SStore.constant_store 0.

  Notation "'s_0'" := zero_store : store_scope.

  (*
     Symbolic state
  *)

  Inductive symbolic_state : Set :=
    SymbolicState : SStore.t → SHeap.t → Formula.t → symbolic_state.

  Notation "⟨ s , h , ϕ ⟩" := (SymbolicState s h ϕ) (at level 0).

  Definition state : field symbolic_state symbolic_state :=
    ((fun σ => σ), (fun σ σ' => σ')).

  Definition store : field symbolic_state SStore.t :=
    (fun σ => match σ with
                | ⟨s, _, _⟩ => s
              end,
     fun σ s => match σ with
                  | ⟨_, h, ϕ⟩ => ⟨s, h, ϕ⟩
                end).

  Definition heap : field symbolic_state SHeap.t :=
    (fun σ => match σ with
                | ⟨_, h, _⟩ => h
              end,
     fun σ h => match σ with
                  | ⟨s, _, ϕ⟩ => ⟨s, h, ϕ⟩
                end).

  Definition path_condition : field symbolic_state Formula.t :=
    (fun σ => match σ with
                | ⟨_, _, ϕ⟩ => ϕ
              end,
     fun σ ϕ => match σ with
                  | ⟨s, h, _⟩ => ⟨s, h, ϕ⟩
                end).

  Definition symbols (σ : symbolic_state) : list Symb.t :=
    match σ with
      | ⟨s, h, Φ⟩ =>
        concat_map Term.symbols (SStore.values s) ++
        concat_map Term.symbols (concat_map SChunk.args (SHeap.enum h)) ++
        Formula.symbols Φ
    end.

  Module SMTImpl := SMT.MakeClueful.

  Definition smt_result := SMT.result.

  Definition decide_sat := SMTImpl.decide.
  Definition derive_value := SMTImpl.get_value.

  Definition smt_result_beq := SMT.result_beq.

  Definition Unsat := SMT.Unsat.

  Open Scope op_scope.

  Definition s_operator := @operator symbolic_state symbolic_state.

  Definition read_store (id : Id.t) : s_operator Term.t :=
    with_current store (SStore.lookup id).

  Definition update_store (id : Id.t) (t : Term.t) : s_operator unit :=
    update_current store (SStore.bind id t).

  Definition evaluate (e : Expression.t) : s_operator Term.t :=
    with_current store (Term.of_expression e ∘ flip SStore.lookup).

  Definition to_formula (b : BooleanExpression.t) : s_operator Formula.t :=
    with_current store (Formula.of_boolean_expression b ∘ flip SStore.lookup).

  Definition with_store
    {A : Set} (s : SStore.t) (op : s_operator A) : s_operator A :=
    s’ ← current store;
    _  ← set_current store s;
    r  ← op;
    _  ← set_current store s’;
    yield r.

  Definition smt (f : Formula.t) : s_operator smt_result := yield (decide_sat f).

  Definition assume_formula (φ : Formula.t) : s_operator unit :=
    Φ ← current path_condition;
    r ← smt (φ && Φ)%fla;
    assume (negb (smt_result_beq r Unsat));
    set_current path_condition (φ && Φ)%fla.

  Definition assume_bexpr (b : BooleanExpression.t) : s_operator unit :=
    φ ← with_current store
                     (Formula.of_boolean_expression b ∘ flip SStore.lookup);
    assume_formula φ.

  Definition assert_formula (φ : Formula.t) : s_operator unit :=
    Φ ← current path_condition;
    r ← smt (Formula.fla_not (Formula.implies Φ φ));
    assert (smt_result_beq r Unsat).

  Definition assert_bexpr (b : BooleanExpression.t) : s_operator unit :=
    φ ← to_formula b; assert_formula φ.

  Definition produce_chunk (α : SChunk.t) : s_operator unit :=
    update_current heap (SHeap.produce α).

  Definition consume_chunk (α : SChunk.t) : s_operator unit :=
    r ← with_current heap (SHeap.consume α);
    match r with
      | Some h’ => set_current heap h’
      | None    => failure
    end.
```

```
Definition pick_chunk_angelically : s_operator SChunk.t :=
  αs ← with_current heap SHeap.enum;
  pick_angelically αs.

Definition find_chunk
  (pred : Predicate.t) (t : Term.t) : s_operator SChunk.t :=
  α ← pick_chunk_angelically;
  match α with
    | SChunk.Chunk p x y => assert (Predicate.beq p pred);
                            assert_formula (x == t)%fla;
                            yield (SChunk.Chunk p x y)
  end.

Definition read_cell (ℓ : Term.t) : s_operator Term.t :=
  α ← find_chunk Predicate.ptr ℓ;
  match α with
    | SChunk.Chunk _ _ v => yield v
  end.

Definition write_cell (ℓ v : Term.t) : s_operator unit :=
  α ← find_chunk Predicate.ptr ℓ;
  consume_chunk α;
  produce_chunk (ℓ ↦ v).

Definition clear_heap : s_operator unit :=
  set_current heap SHeap.empty.

Definition leak_check : s_operator unit :=
  r ← with_current heap SHeap.is_empty; assert r.

Definition fresh_symbol (exclude : list Symb.t) : s_operator Symb.t :=
  let ξ := (proj1_sig (Symb.fresh exclude))
  in
  assume_formula (Formula.fla_eq (Term.term_sym ξ) (Term.term_sym ξ));
  yield ξ.

Definition fresh_symbol_n (n : nat) : s_operator (list Symb.t) :=
  let aux  :=
    fix rec (n : nat) (exclude : list Symb.t) : s_operator (list Symb.t) :=
    match n with
      | O   => yield nil
      | S n => ξ ← fresh_symbol exclude;
               ξs ← rec n (ξ :: exclude);
               yield (ξ :: ξs)
    end
    in
    σ ← current state;
    let exclude := symbols σ
    in
    aux n exclude.

Definition fresh_symbol_1 : s_operator Symb.t :=
  ξs ← fresh_symbol_n 1;
  match ξs with
    | ξ :: nil => yield ξ
    | _        => failure
  end.
```

```
Definition allocate (n : nat) : s_operator Term.t :=
  let allocate_at :=
    fix aux (ℓ : Symb.t) (vs : list Symb.t) : s_operator unit :=
    match vs with
      | nil   => nop
      | v::vs =>
        let k := List.length vs
        in
        let α := ((Term.term_sym ℓ + k)%term ↦ (Term.term_sym v))%chunk
        in
        produce_chunk α;
        aux ℓ vs
    end
  in
  ξs ← fresh_symbol_n (S n);
  match ξs with
    | ℓ::vs => allocate_at ℓ vs;
               produce_chunk (mb [(Term.term_sym ℓ); n])%chunk;
               yield (Term.term_sym ℓ)
    | nil   => failure
  end.

Fixpoint consume_cells (ℓ : Term.t) (n : nat) : s_operator unit :=
  match n with
    | O   =>
      nop
    | S n =>
      α ← find_chunk Predicate.ptr (ℓ + n);
      consume_chunk α;
      consume_cells ℓ n
  end.

Definition block_size (ℓ : Term.t) : s_operator nat :=
  α ← find_chunk Predicate.mb ℓ;
  match α with
    | SChunk.Chunk _ _ t =>
      Φ ← current path_condition;
      match derive_value Φ t with
        | Some n => yield n
        | None   => failure
      end
  end.

Definition consume_mb (ℓ : Term.t) : s_operator nat :=
  α ← find_chunk Predicate.mb ℓ;
  consume_chunk α;
  match α with
    | SChunk.Chunk _ _ t =>
      Φ ← current path_condition;
      match derive_value Φ t with
        | Some n => yield n
        | None   => failure
      end
  end.
```

```
Fixpoint consume_assertion (a : Assertion.t) : s_operator unit :=
  match a with
    | Assertion.bexpr b =>
      assert_bexpr b
    | Assertion.sepand a1 a2 =>
      consume_assertion a1;
      consume_assertion a2
    | Assertion.cond b a1 a2 =>
      let then_op := assume_bexpr b; consume_assertion a1 in
      let else_op := assume_bexpr (¬ b)%bexpr; consume_assertion a2 in
        op ← pick_demonically (then_op :: else_op :: nil);
        op
    | Assertion.pred p e x =>
      t ← evaluate e;
      α ← find_chunk p t;
      consume_chunk α;
      match α with
        | SChunk.Chunk p t1 t2 => update_store x t2
      end
  end.

Fixpoint produce_assertion (a : Assertion.t) : s_operator unit :=
  match a with
    | Assertion.bexpr b =>
      assume_bexpr b
    | Assertion.sepand a1 a2 =>
      produce_assertion a1;
      produce_assertion a2
    | Assertion.cond b a1 a2 =>
      let then_op  := assume_bexpr b; produce_assertion a1 in
      let else_op := assume_bexpr (¬ b)%bexpr; produce_assertion a2 in
        op ← pick_demonically (then_op :: else_op :: nil);
        op
    | Assertion.pred p e x =>
      t1 ← evaluate e;
      ξ ← fresh_symbol_1;
      let t2 := Term.term_sym ξ in
      produce_chunk (SChunk.Chunk p t1 t2);
      update_store x t2
  end.

Section WithProgram.
  Variable (Π : SILPP.program).

  Open Scope command_scope.

  Fixpoint symbolic_execution (c : SILPP.command) : s_operator unit :=
    match c with
      | skip => yield □
      | x := e =>
        v ← evaluate e;
        update_store x v
      | c; c' =>
        symbolic_execution c;
        symbolic_execution c'
```

```
| x := malloc n =>
  ℓ ← allocate n;
  update_store x ℓ
| free e =>
  ℓ ← evaluate e;
  n ← consume_mb ℓ;
  consume_cells ℓ n
| ⟦e⟧ := e' =>
  ℓ ← evaluate e;
  v ← evaluate e';
  write_cell ℓ v
| x := ⟦e⟧ =>
  ℓ ← evaluate e;
  v ← read_cell ℓ;
  update_store x v
| If b Then c Else c' =>
  let then_clause :=
    assume_bexpr b;
    symbolic_execution c
  in
  let else_clause :=
    assume_bexpr (¬ b)%bexpr;
    symbolic_execution c'
  in
  let clauses := then_clause :: else_clause :: nil
  in
  op ← pick_demonically clauses;
  op
| open p[e; ?] =>
  ℓ ← evaluate e;
  α ← find_chunk p ℓ;
  _ ← consume_chunk α;
  preddef ← from_some (SILPP.predicates Π p);
  let x := SILPP.argument_a preddef
  in
  let y := SILPP.argument_b preddef
  in
  let body := SILPP.predicate_body preddef in
    match α with
      | SChunk.Chunk _ ℓ' v' =>
      with_store (s_0⟦x := ℓ'⟧⟦y := v'⟧)%store
                 (produce_assertion body)
    end
| close p[e; e'] =>
  ℓ  ← evaluate e;
  v ← evaluate e';
  preddef ← from_some (SILPP.predicates Π p);
  let x := SILPP.argument_a preddef in
  let y := SILPP.argument_b preddef in
  let body := SILPP.predicate_body preddef in
  with_store (s_0⟦x := ℓ⟧⟦y := v⟧)%store
             (consume_assertion body);
  produce_chunk (SChunk.Chunk p ℓ v)
```

```
        | r[e] =>
          rdef ← from_some (SILPP.routines Π r);
          let x := SILPP.argument rdef in
          t ← evaluate e;
          with_store (s_0⟦x := t⟧)%store
                     (consume_assertion (SILPP.precondition rdef);
                      produce_assertion (SILPP.postcondition rdef))
      end.

    Close Scope command_scope.

    Definition valid_routine (rdef : SILPP.routine_definition) : bool :=
      let validation_op :=
        ξ ← fresh_symbol_1;
        let v := Term.term_sym ξ in
        update_store (SILPP.argument rdef) v;
        produce_assertion (SILPP.precondition rdef);
        with_store (s_0⟦(SILPP.argument rdef) := v⟧)%store
                   (symbolic_execution (SILPP.routine_body rdef));
        consume_assertion (SILPP.postcondition rdef);
        leak_check
      in
      let pc := Formula.fla_eq (Term.term_lit 0) (Term.term_lit 0)
      in
      let σ := ⟨s_0, SHeap.empty, pc ⟩
      in
        negb (is_bottom (validation_op σ)).

  End WithProgram.

  Close Scope op_scope.

End Make.
```

D.13 EWP

```
(**

  IMPORTANT: This proof script was developed with Coq 8.1.
  It will NOT work with Coq 8.2 or later
  (due to differences in the standard library).

  The script contains four Coq axioms (search for "Axiom").
  One is borrowed from Coq 8.2's standard library, namely
  [functional_extensionality_dep]. The others are directly
  related to the paper and are trivial (but necessary).
  All proofs are complete, none has been ended
  with "Admitted" (which would tell Coq to just
  accept them as axioms).

  The validity of the entire Coq script depends on a small
  core of definitions. An error here could make
  the entire proof worthless. We identify these
  "Achilles heel spots" with a clear message. Fortunately,
  many of these "fragile definitions" are trivial.

*)

(*

  TOC
  --
  1. CUSTOM TACTICS
  2. GENERAL DEFINITIONS
  3. PAPER SPECIFIC
  3.1. LANGUAGE DEFINITIONS AND THEOREMS
  3.2. SINGLE ASSIGNMENT
  3.3. PASSIFICATION
  3.4. WEAKEST PRECONDITIONS SOUNDNESS
  3.5. WEAKEST PRECONDITIONS SIZE

*)

Require Import Arith.
Require Import Omega.
Require Import FSets.
Require Import Max.
Require Import List.
Require Import Relations.
Require Import Setoid.
Require Import ProofIrrelevance.

Set Printing Width 200.
```

```
(** * Custom Tactics *)

(**
   This tactic introduces a new identifier [id], which is equal to [term].
«

    =====================
     Goal

introduce x (3 + 5)

    x : nat
    H : x = 3 + 5
    =====================
     Goal
»

    Generally useful to simplify expressions by substituting entire
    subexpressions with a single identifier (using [rewrite]), or to
    apply the [induction] tactic which sometimes tends to throw some
    information away.

*)
Ltac introduce_eq id term :=
  ( set (id := term);
    assert (id = term);
    [ trivial | clearbody id ]).

(**
   New tactic notation: instead of [introduce_eq x t]
   we can write [introduce new identifier x for t].
*)
Tactic Notation "introduce" "new" "identifier" ident(x) "for" constr(t) :=
  introduce_eq x t.

(**
   Same as [introduce_eq], but also performs a rewrite in [H].

«
  x : nat
  y : nat
  z : nat
  H : x = (3 + y) * z
  ===========================
   Goal

introduce_eq_in a (3 + y) H.

  x : nat
  y : nat
  z : nat
  a : nat
  H0 : a = 3 + y
  H : x = a * z
  ===========================
   Goal
» *)
```

```
Ltac introduce_eq_in id term H :=
  ( let H' := fresh in
      set (id := term);
      assert (H' : id = term);
      [ trivial | clearbody id ];
      rewrite <- H' in H ).

(**
   Adds a new hypothesis, whose proof is given by [term].

«
  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  ============================
   Goal

state_fresh (lt_trans _ _ _ H H0).

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : a < c
  ============================
   Goal
» *)
Ltac state_fresh term :=
  let id := fresh in
    (assert (id := term)).

(**
   Same as [state_fresh], but instead of using a fresh id, it uses [H].

«
  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  ============================
   Goal

state_as Foo (lt_trans _ _ _ H H0).

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  Foo : a < c
  ============================
   Goal
» *)
```

```
Ltac state_as H term :=
  (assert (H := term)).

(**
   Introduces a new tactic notation:
   [state_fresh term] can now be written [state term].
*)
Tactic Notation "state" constr(x) :=
  state_fresh x.

(**
   Introduces a new tactic notation:
   [state_as H term] can now be written [state term as H].
*)
Tactic Notation "state" constr(x) "as" ident(H) :=
  state_as H x.

(**
   Applies [x] to [H].

«
  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  ============================
   Goal

state lt_trans.

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : forall n m p : nat, n < m -> m < p -> n < p
  ============================
   Goal

specify_single H1 a.

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : forall m p : nat, a < m -> m < p -> a < p
  ============================
   Goal
specify_single H1 b.

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : forall p : nat, a < b -> b < p -> a < p
  ============================
   Goal

specify_single H1 c.

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : a < b -> b < c -> a < c
  ============================
   Goal

specify_single H1 H.

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : b < c -> a < c
  ============================
   Goal

specify_single H1 H0.

  a : nat
  b : nat
  c : nat
  H : a < b
  H0 : b < c
  H1 : a < c
  ============================
   Goal
»

  Useful in keeping hypotheses "up-to-date" with the proof developments.
  Also simplifies things a bit when the terms applied are large.
*)
Ltac specify_single H x :=
  (let H' := fresh in
    state (H x) as H';
    clear H;
    rename H' into H).
```

```
Tactic Notation "specify" hyp(H) constr(x1) :=
  specify_single H x1.

Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2) :=
  specify_single H x1;
  specify_single H x2.

Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3.

Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4.

Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4)
                                 constr(x5) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4;
  specify_single H x5.

Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4)
                                 constr(x5)
                                 constr(x6) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4;
  specify_single H x5;
  specify_single H x6.

Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4)
                                 constr(x5)
                                 constr(x6)
                                 constr(x7) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4;
  specify_single H x5;
  specify_single H x6;
  specify_single H x7.
Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4)
                                 constr(x5)
                                 constr(x6)
                                 constr(x7)
                                 constr(x8) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4;
  specify_single H x5;
  specify_single H x6;
  specify_single H x7;
  specify_single H x8.
Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4)
                                 constr(x5)
                                 constr(x6)
                                 constr(x7)
                                 constr(x8)
                                 constr(x9) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4;
  specify_single H x5;
  specify_single H x6;
  specify_single H x7;
  specify_single H x8;
  specify_single H x9.
Tactic Notation "specify" hyp(H) constr(x1)
                                 constr(x2)
                                 constr(x3)
                                 constr(x4)
                                 constr(x5)
                                 constr(x6)
                                 constr(x7)
                                 constr(x8)
                                 constr(x9)
                                 constr(x10) :=
  specify_single H x1;
  specify_single H x2;
  specify_single H x3;
  specify_single H x4;
  specify_single H x5;
  specify_single H x6;
  specify_single H x7;
  specify_single H x8;
  specify_single H x9;
  specify_single H x10.

(**
   Fancy [injection] tactic, which also introduces the equalities
   produced by [injection] as hypotheses and performs substitutions.

«
  a : nat
  b : nat
  c : nat
  H : S a = S b
  H0 : a = c
  ============================
   b = c

strip H.

  b : nat
  c : nat
  H : c = b
  ============================
   b = c
»*)
Ltac strip h :=
  (injection h; clear h; intros; subst).

(**
   Breaks open a pair [p] and names the components [x] and [y].
   Used to call a function returning two values as pair and binding
   both values to identifiers in one step.
*)
Ltac introduce_pair p x y :=
  (let z := fresh in introduce_eq z p;
                   destruct z as [ x y ]).

(**
   New tactic notation: [introduce_pair p x y] can be
   written as [introduce pair p as x y].
*)
Tactic Notation "introduce" "pair" constr(p) "as" ident(x) ident(y) :=
  (introduce_pair p x y).
```

```
(**
   Deals with an existential in a hypothesis.

«
  H : exists n : nat, forall k : nat, n > k
  ============================
   False

elim_ex H x.

  x : nat
  H : forall k : nat, x > k
  ============================
   False
»*)
Ltac elim_ex H n :=
  let H' := fresh in elim H;
                     intros n H';
                     clear H;
                     rename H' into H.

(**
   New tactic notation: [elim_ex H x] can be written
   [eliminate existential x in H].
   We also define notations to get rid of multiple existentials in one step.
*)
Tactic Notation "eliminate" "existential" ident(x) "in" hyp(H) :=
  elim_ex H x.

Tactic Notation "eliminate" "existentials" ident(x) ident(y) "in" hyp(H) :=
  elim_ex H x; elim_ex H y.

Tactic Notation "eliminate" "existentials" ident(x) ident(y) ident(z)
                "in" hyp(H) :=
  elim_ex H x; elim_ex H y; elim_ex H z.

Tactic Notation "eliminate" "existential" ident(x) "in" hyp(H) "as" ident(H') :=
  elim_ex H x; rename H into H'.

Tactic Notation "eliminate" "existentials" ident(x) ident(y) "in"
                hyp(H) "as" ident(H') :=
  elim_ex H x; elim_ex H y; rename H into H'.

Tactic Notation
  "eliminate" "existentials" ident(x) ident(y) ident(z)
  "in" hyp(H) "as" ident(H') :=
    elim_ex H x; elim_ex H y; elim_ex H z; rename H into H'.

(**

   Used where an inequality needs to be proved.
   Can refine both left and right bound.
   Refining the left bound:

«
                             X <= Y
   X <= Y                    Y <= Z
   ------   ==>  ------  /\   ------
   X <= Z        Y <= Z       X <= Z
»
   where the right goal is proved automatically using the omega tactic.

   Refining the right bound:
«
                             X <= Y
   Y <= Z                    Y <= Z
   ------   ==>  ------  /\   ------
   X <= Z        X <= Y       X <= Z
»
   where the right goal is proved automatically using the omega tactic.

*)
Ltac refine_le H :=
  match goal with
    | [ H : le ?X ?Y |- le ?X ?Z ] => cut (Y <= Z); [ intros; omega | idtac ]
    | [ H : le ?Y ?Z |- le ?X ?Z ] => cut (X <= Y); [ intros; omega | idtac ]
  end.

(**
   A tactic to perform an algebraic manipulation. [algebraic_rewrite x y] first
   needs to prove that [x] and [y] are equivalent,
   which it tries to do automatically
   (if this fails, a message is printed and the proof is left to the user).
   Next, it also rewrites [x] as [y] in all hypotheses and the goal.

«
  a : nat
  b : nat
  ============================
   (a + b) * (a + b) >= a * a

algebraic_rewrite ((a + b) * (a + b)) (a * a + 2 * a * b + b * b).

  a : nat
  b : nat
  ============================
   a * a + 2 * a * b + b * b >= a * a
» *)
Ltac algebraic_rewrite x y :=
  let H := fresh in
    assert (H : x = y);
      [ try ring;
        idtac "Failed to automatically prove rewrite (not an error)"
        | rewrite H in * |- *;
          clear H ].
```

```
(** New tactic notation: [algebraic_rewrite x y] can be
    written as [algebraically rewrite x as y]. *)
Tactic Notation "algebraically" "rewrite" constr(x) "as" constr(y) :=
  algebraic_rewrite x y.

(**
   All-in-one tactic to solve nat-related goals.
*)
Ltac solveq :=
  simpl in * |- *; solve [ auto with arith | ring | omega ].

(**
   A tactic which helps with proving inequalities.
   Used to bring the lower bound closer to the upper bound.

«
  a : nat
  b : nat
  ============================
   a - b <= a + 5

refine_le_left a.

  a : nat
  b : nat
  ============================
   a <= a + 5
» *)
Ltac refine_le_left Y :=
  match goal with
    | |- le ?X ?Z =>
      let H := fresh in
        assert (H : X <= Y);
        [ try solveq; idtac "Failed to refine automatically (not an error)"
        | refine_le H; clear H ]
  end.

(**
   A tactic which helps with proving inequalities.
   Used to bring the upper bound closer to the lower bound.

«
  a : nat
  b : nat
  ============================
   a - b <= a + 5

refine_le_right a.

  a : nat
  b : nat
  ============================
   a - b <= a
» *)
```

```
Ltac refine_le_right Y :=
  match goal with
    | |- le ?X ?Z =>
      let H := fresh in
        assert (H : Y <= Z);
        [ try solveq; idtac "Failed to refine automatically (not an error)"
        | refine_le H; clear H ]
  end.

(**
   New tactic notation: [refine_le_left x]
   can be written [refine left bound with x].
*)
Tactic Notation "refine" "left" "bound" "with" constr(x) :=
  refine_le_left x.

(**
   New tactic notation: [refine_le_right x]
   can be written [refine right bound with x].
*)
Tactic Notation "refine" "right" "bound" "with" constr(x) :=
  refine_le_right x.

(** * General definitions *)

(** ** Decidability *)

(**
   We define decidability as a Set, so that we can use it in algorithms.
*)
Definition decidable (P : Prop) := {P} + {~P}.

(**
   [decidable_eq A] means that we can decide whether
   two values of type [A] are equal or not.
*)
Definition decidable_eq (A : Set) :=
  forall x y : A, decidable (x = y).

(** ** Equivalent functions *)

(**
   We say two functions [f] and [g] with domain [A]
   are equivalent when [f x = g x], forall [x] in [A].
   Function equivalence is an equivalence relation,
   as will be proved in later theorems.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition function (A B : Type) :=
  A -> B.

(**
   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
```

```
Definition equivalent_functions (A B : Type) (f g : function A B) :=
  forall (x : A), f x = g x.

Implicit Arguments equivalent_functions [ A B ].

(**
   We show a function is equivalent with itself (reflexivity).
*)
Theorem equivalent_functions_refl : forall A B f,
  (@equivalent_functions A B f f).
Proof. (* 1 lines *) Qed.

(**
   We show that if [f] is equivalent with [g] and
   [g] is equivalent with [h],
   then [f] is equivalent with [h] (transitivity).
*)
Theorem equivalent_functions_trans : forall A B f g h,
  (@equivalent_functions A B f g) ->
  (@equivalent_functions A B g h) ->
  (@equivalent_functions A B f h).
Proof. (* 2 lines *) Qed.

(**
   We show that if [f] is equivalent with [g],
   then [g] is equivalent with [f] (symmetricity).
*)
Theorem equivalent_functions_symm : forall A B f g,
  (@equivalent_functions A B f g) ->
  (@equivalent_functions A B g f).
Proof. (* 2 lines *) Qed.

Implicit Arguments equivalent_functions_refl [ A B ].

Implicit Arguments equivalent_functions_trans [ A B ].

Implicit Arguments equivalent_functions_trans [ A B ].

(**
   We put these theorems in the hint database so
   that we can use the [auto] tactic to easily prove
   goals that rely on these properties.
*)
Hint Resolve equivalent_functions_refl.

Hint Resolve equivalent_functions_symm.

Add Relation function equivalent_functions
  reflexivity proved by equivalent_functions_refl
  symmetry proved by equivalent_functions_symm
  transitivity proved by equivalent_functions_trans
as equivalent_functions_rel.

(**
   Functional extensionality. Taken from Coq 8.2's standard library.
*)
Axiom functional_extensionality_dep : forall A (B : A -> Type),
  forall (f g : forall x : A, B x), (forall x, f x = g x) -> f = g.
```

```
Lemma functional_extensionality A B (f g : A -> B) :
  (forall x, f x = g x) -> f = g.
Proof. (* 3 lines *) Qed.

Lemma equal_f :
  forall (A B : Type) (f g : A -> B),
    f = g -> forall x, f x = g x.
Proof. (* 3 lines *) Qed.

Lemma eta_expansion_dep A (B : A -> Type) (f : forall x : A, B x) :
  f = fun x => f x.
Proof. (* 3 lines *) Qed.

Lemma eta_expansion A B (f : A -> B) : f = fun x => f x.
Proof. (* 2 lines *) Qed.

(** * Paper specific *)

(** ** Language definitions and theorems *)

(**
   We define identifiers as natural numbers ([nat]).
   We don't rely on this internal representation
   (we make [id] opaque a bit later)
   except for the [id]-set definitions, for which
   we need an ordered type; for this reason, [nat]
   seemed like a natural choice.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition id := nat.

(**
   A vid (versioned id) is an id with a version number.
   The [%type] suffix tells Coq it has to
   evaluate [(id * nat)] in the type scope,
   otherwise it will interpret [*] as [nat]-multiplication.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition vid := (id * nat)%type.

(**
   We define id as an ordered type (will be used
   later to define identifier-sets).
*)
Module Identifier_OT <: OrderedType.

  Definition t := id.

  Definition eq (x y : t) := x = y.

  Definition lt (x y : t) := x < y.

  Theorem eq_refl : forall x, eq x x.
  Proof. (* 1 lines *) Qed.
  Theorem eq_sym : forall x y : t, eq x y -> eq y x.
    unfold eq; auto.
  Qed.

  Theorem eq_trans : forall x y z, eq x y -> eq y z -> eq x z.
  Proof. (* 1 lines *) Qed.

  Definition lt_trans := lt_trans.

  Theorem lt_not_eq : forall x y, lt x y -> ~ eq x y.
  Proof. (* 6 lines *) Qed.

  Definition compare : forall x y, Compare lt eq x y.
  Proof. (* 5 lines *) Qed.

End Identifier_OT.

(** We define sets of identifiers *)
Module IdSet := FSetList.Make(Identifier_OT).

(**
   As promised earlier, we make id opaque, making sure we don't make use
   of its internal implementation.
*)
Opaque id.

(** Generate a few extra theorems for IdSets. *)
Module IdSetProperties := Properties ( IdSet ).

(**
   If #(X &cup; Y) &sube; Z#, then #X &sube; Z# and #Y &sube; Z#.
*)
Lemma union_subset :
  forall x y z,
    IdSet.Subset (IdSet.union x y) z ->
    IdSet.Subset x z /\ IdSet.Subset y z.
Proof. (* 4 lines *) Qed.

(**
   Decidability theorems stated as a definition
   to make it transparent (needed for the single
   assignment algorithm to be "fully evaluateable").

   This theorem states that we can decide whether two [id]s are equal or not.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition decidable_eq_id : decidable_eq id.
  unfold decidable_eq; unfold decidable; intros.
  apply (eq_nat_dec x y).
Defined.

(**
   States that we can decide whether two [vid]s are equal or not.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
```

```
Definition decidable_eq_vid : decidable_eq vid.
  unfold decidable_eq; unfold decidable; unfold vid; intros.
  destruct x; destruct y.
  rename i0 into j; rename n0 into m.
  destruct (decidable_eq_id i j);
    destruct (eq_nat_dec n m);
      try (left; subst; trivial; fail);
        right; red; intros; injection H; intros; contradiction.
Defined.

(**
   We define the set of values as well
   as two elements: T (true) and
   "some other value" F which is not equal to true.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Parameters
  (value : Set)
  (T     : value)
  (F     : value).

(**
   We state we can check whether two [value]s are equal.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Axiom decidable_eq_value : decidable_eq value.

(**
   We have defined the existence of two [value]s,
   [T] and [F]. We define them to be unequal.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Axiom T_neq_F : T <> F.

(**
   A [store] is a total mapping from [id]s to [value]s.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition store  := id -> value.

(**
   A [vstore] is a total mapping from [id]s to [value]s.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition vstore := vid -> value.
```

```
(**
   A [vmap] (short for version map) maps non-versioned
   identifiers ([vid]s) to a version number ([nat]).

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition vmap := id -> nat.

(**
   An expression is a total mapping from [store]s to [value]s.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition expr  := store -> value.
Definition vexpr := vstore -> value.

(**
   The original set of commands.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive command : Set :=
| cAssert   : expr -> command
| cAssume   : expr -> command
| cAssign   : id -> expr -> command
| cSequence : command -> command -> command
| cSkip     : command
| cChoice   : command -> command -> command.

(**
   The versioned set of commands.
   Will be produced by the [transform_sa] algorithm.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive vcommand : Set :=
| vcAssert   : vexpr -> vcommand
| vcAssume   : vexpr -> vcommand
| vcAssign   : vid -> vexpr -> vcommand
| vcSequence : vcommand -> vcommand -> vcommand
| vcSkip     : vcommand
| vcChoice   : vcommand -> vcommand -> vcommand.

(**
   The versioned set of commands, without assignment
   Will be produced by the [passify] algorithm.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
```

```
Inductive pcommand : Set :=
| pcAssert   : vexpr -> pcommand
| pcAssume   : vexpr -> pcommand
| pcSequence : pcommand -> pcommand -> pcommand
| pcSkip     : pcommand
| pcChoice   : pcommand -> pcommand -> pcommand.

(**
   Metric on commands.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Fixpoint command_metric (c : command) : nat :=
  match c with
    | cAssert _     => 2
    | cAssume _     => 2
    | cAssign _ _   => 2
    | cSequence x y => S (command_metric x + command_metric y)
    | cSkip         => 1
    | cChoice x y   => S (command_metric x + command_metric y)
  end.

(**
   Metric on vcommands.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Fixpoint vcommand_metric (c : vcommand) : nat :=
  match c with
    | vcAssert _     => 2
    | vcAssume _     => 2
    | vcAssign _ _   => 2
    | vcSequence x y => S (vcommand_metric x + vcommand_metric y)
    | vcSkip         => 1
    | vcChoice x y   => S (vcommand_metric x + vcommand_metric y)
  end.

(**
   Metric on pcommands.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Fixpoint pcommand_metric (c : pcommand) : nat :=
  match c with
    | pcAssert _     => 2
    | pcAssume _     => 2
    | pcSequence x y => S (pcommand_metric x + pcommand_metric y)
    | pcSkip         => 1
    | pcChoice x y   => S (pcommand_metric x + pcommand_metric y)
  end.
```

```
(**
   Create a versioned expression of a command.
   E.g. the expression "x == y + 1" will be transformed to
   "x_5 == y_3 + 1" under the version map { x -> 5, y -> 3 }.
*)
Definition version_expr (e : expr) (v : vmap) : vexpr :=
  fun (vmu : vstore) => e (fun x => vmu (x, v x)).

(**
   Modifies a single mapping of a total function.
«
   (rebind f x y) x  = y
   (rebind f x y) x' = f x'    with x <> x'
»

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition rebind (A B : Set)
                  (eq_dec : decidable_eq A)
                  (f : A -> B)
                  (x : A)
                  (y : B) :=
  fun a => if eq_dec a x then y else f a.

Implicit Arguments rebind [ A B ].

(**
   Increment the version of an identifier.

   E.g.
«
   inc { x -> 5, y -> 3, ... } x = { x -> 6, y -> 3, ... }
» *)
Definition inc (v : vmap) (x : id) :=
  rebind decidable_eq_id v x (S (v x)).

(**
   Updates a store binding.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition update_store (mu : store) (x : id) (v : value) :=
  rebind decidable_eq_id mu x v.

(**
   Update a versioned store binding.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Definition update_vstore (mu : vstore) (x : vid) (v : value) :=
  rebind decidable_eq_vid mu x v.
```

```
(**
   Checks if a regular store and a versioned store
   are equivalent under a certain version map.

   E.g. the regular store
        { x -> 12, y -> 44 }
        is equivalent with the versioned store
        { x_0 -> -5, x_1 -> 2, x_2 -> 12, y_0 -> 44 }
        under the version map { x -> 2, y -> 0 }
*)
Definition store_sync_vstore (mu : store) (v : vmap) (vmu : vstore) :=
  equivalent_functions mu (fun x => vmu (x, v x)).

(**
   Two expression evaluate to the same value
   under equivalent stores.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Axiom expression_evaluation : forall (e : expr) mu mu',
  equivalent_functions mu mu' -> e mu = e mu'.

Theorem sync_stores : forall mu v vmu (e : expr),
  store_sync_vstore mu v vmu -> e mu = ((version_expr e v) vmu).
Proof. (* 5 lines *) Qed.

(**
   States for the original command language.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive state : Set :=
| ip       : command -> store -> state
| failure  : store -> state.

(**
   States for the single assignment phase.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive vstate : Set :=
| vip       : vcommand -> vstore -> vstate
| vfailure  : vstore -> vstate.

(**
   States for the passification phase.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive pstate : Set :=
| pip       : pcommand -> pstate
| pfailure  : pstate.
```

```
(**
   Metric on states
*)
Definition state_metric (s : state) : nat :=
  match s with
    | ip c _    => command_metric c
    | failure _ => 0
  end.

(**
   Metric on vstates
*)
Definition vstate_metric (s : vstate) : nat :=
  match s with
    | vip c _    => vcommand_metric c
    | vfailure _ => 0
  end.

(**
   Metric on pstates
*)
Definition pstate_metric (s : pstate) : nat :=
  match s with
    | pip c    => pcommand_metric c
    | pfailure => 0
  end.

(**
   Regular single step operational semantics.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive step : state -> state -> Prop :=
| stepAssertT : forall e mu,
    e mu = T ->
    step (ip (cAssert e) mu) (ip cSkip mu)

| stepAssertF : forall e mu,
    e mu <> T ->
    step (ip (cAssert e) mu) (failure mu)

| stepAssume : forall e mu,
    e mu = T ->
    step (ip (cAssume e) mu) (ip cSkip mu)

| stepSeq : forall c1 c1' c2 mu mu',
    step (ip c1 mu) (ip c1' mu') ->
    step (ip (cSequence c1 c2) mu)
         (ip (cSequence c1' c2) mu')

| stepSeqSkip : forall c2 mu,
    step (ip (cSequence cSkip c2) mu) (ip c2 mu)

| stepSeqFail : forall c1 c2 mu mu',
    step (ip c1 mu) (failure mu') ->
    step (ip (cSequence c1 c2) mu) (failure mu')
```

```
| stepAssign : forall x e mu,
    step (ip (cAssign x e) mu)
         (ip cSkip (update_store mu x (e mu)))

| stepChoiceL : forall c1 c2 mu,
    step (ip (cChoice c1 c2) mu)
         (ip c1 mu)

| stepChoiceR : forall c1 c2 mu,
    step (ip (cChoice c1 c2) mu)
         (ip c2 mu).

(**
   Versioned single step operational semantics.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive vstep : vstate -> vstate -> Prop :=
| vstepAssertT : forall e vmu,
    e vmu = T ->
    vstep (vip (vcAssert e) vmu) (vip vcSkip vmu)

| vstepAssertF : forall e vmu,
    e vmu <> T ->
    vstep (vip (vcAssert e) vmu) (vfailure vmu)

| vstepAssume  : forall e vmu, e vmu = T ->
    vstep (vip (vcAssume e) vmu) (vip vcSkip vmu)

| vstepSeq     : forall c1 c1' c2 vmu vmu',
    vstep (vip c1 vmu) (vip c1' vmu') ->
    vstep (vip (vcSequence c1 c2) vmu)
          (vip (vcSequence c1' c2) vmu')

| vstepSeqSkip : forall c2 vmu,
    vstep (vip (vcSequence vcSkip c2) vmu) (vip c2 vmu)

| vstepSeqFail : forall c1 c2 vmu vmu',
    vstep (vip c1 vmu) (vfailure vmu') ->
    vstep (vip (vcSequence c1 c2) vmu) (vfailure vmu')

| vstepAssign  : forall x e vmu,
    vstep (vip (vcAssign x e) vmu)
          (vip vcSkip (update_vstore vmu x (e vmu)))

| vstepChoiceL : forall c1 c2 vmu,
    vstep (vip (vcChoice c1 c2) vmu) (vip c1 vmu)

| vstepChoiceR : forall c1 c2 vmu,
    vstep (vip (vcChoice c1 c2) vmu) (vip c2 vmu).

(**
   Versioned stateless single step operational semantics.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
```

```
Inductive pstep (vmu : vstore) : pstate -> pstate -> Prop :=
| pstepAssertT : forall e,
    e vmu = T ->
    pstep vmu (pip (pcAssert e)) (pip pcSkip)

| pstepAssertF : forall e,
    e vmu <> T ->
    pstep vmu (pip (pcAssert e)) pfailure

| pstepAssume  : forall e,
    e vmu = T ->
    pstep vmu (pip (pcAssume e)) (pip pcSkip)

| pstepSeq     : forall c1 c1' c2,
    pstep vmu (pip c1) (pip c1') ->
    pstep vmu (pip (pcSequence c1 c2)) (pip (pcSequence c1' c2))

| pstepSeqSkip : forall c2,
    pstep vmu (pip (pcSequence pcSkip c2))
              (pip c2)

| pstepSeqFail : forall c1 c2,
    pstep vmu (pip c1) pfailure ->
    pstep vmu (pip (pcSequence c1 c2)) pfailure

| pstepChoiceL : forall c1 c2,
    pstep vmu (pip (pcChoice c1 c2)) (pip c1)

| pstepChoiceR : forall c1 c2,
    pstep vmu (pip (pcChoice c1 c2)) (pip c2).

(**
   Regular multistep.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive multistep : state -> state -> Prop :=
| multiReflexivity : forall s, multistep s s
| multiStep : forall s1 s2 s3,
    step s1 s2 ->
    multistep s2 s3 ->
    multistep s1 s3.

(**
   Versioned multistep.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive vmultistep : vstate -> vstate -> Prop :=
| vmultiReflexivity : forall s, vmultistep s s
| vmultiStep : forall s1 s2 s3,
    vstep s1 s2 ->
    vmultistep s2 s3 ->
    vmultistep  s1 s3.
```

```
(**
   Versioned stateless multistep.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive pmultistep (vmu : vstore) : pstate -> pstate -> Prop :=
| pmultiReflexivity : forall s, pmultistep vmu s s
| pmultiStep : forall s1 s2 s3,
    pstep vmu s1 s2 ->
    pmultistep vmu s2 s3 ->
    pmultistep vmu s1 s3.

(**
   Helper tactic to perform induction on [step]/[multistep]/[vstep]/[vmultistep].
   Using the [induction] tactic sometimes throws important information away.

   For example, if we don’t use [introduce_step_states], we get
«
  c : command
  c’ : command
  mu : store
  mu’ : store
  H : multistep (ip c mu) (ip c’ mu’)
  ============================
   command_metric c’ <= command_metric c

induction H.

subgoal 1 is:

  c : command
  c’ : command
  mu : store
  mu’ : store
  s : state
  ============================
   command_metric c’ <= command_metric c

subgoal 2 is:

  c : command
  c’ : command
  mu : store
  mu’ : store
  s1 : state
  s2 : state
  s3 : state
  H : step s1 s2
  H0 : multistep s2 s3
  IHmultistep : command_metric c’ <= command_metric c
  ============================
   command_metric c’ <= command_metric c
»
```

```
  As you can see, there is no way of solving subgoal 1:
  the hypotheses do not provide any kind of information
  about either [c] or [c']. We can use
  [introduce_step_states] to preserve this information:

«
  c : command
  c' : command
  mu : store
  mu' : store
  H : multistep (ip c mu) (ip c' mu')
  ============================
   command_metric c' <= command_metric c

introduce_step_states H s s'.

  c : command
  c' : command
  mu : store
  mu' : store
  s : state
  H0 : s = ip c mu
  s' : state
  H1 : s' = ip c' mu'
  H : multistep s s'
  ============================
   command_metric c' <= command_metric c

revert c c' mu mu' H0 H1.

  s : state
  s' : state
  H : multistep s s'
  ============================
   forall (c c' : command) (mu mu' : store),
     s = ip c mu -> s' = ip c' mu' -> command_metric c' <= command_metric c

subgoal 1 is:

  c : command
  c' : command
  mu : store
  mu' : store
  H1 : ip c mu = ip c' mu'
  ============================
   command_metric c' <= command_metric c

subgoal 2 is:

  s2 : state
  c : command
  c' : command
  mu : store
  mu' : store
  H0 : multistep s2 (ip c' mu')
  IHmultistep : forall (c c'0 : command) (mu mu'0 : store),
    s2 = ip c mu -> ip c' mu' = ip c'0 mu'0 ->
```

```
      command_metric c'0 <= command_metric c
  H : step (ip c mu) s2
  ============================
   command_metric c' <= command_metric c
» *)
Ltac introduce_step_states H s s' :=
  match goal with
    | [ H : step ?X ?Y |- _ ]       =>
        introduce_eq_in s X H; introduce_eq_in s' Y H
    | [ H : multistep ?X ?Y |- _ ]  =>
        introduce_eq_in s X H; introduce_eq_in s' Y H
    | [ H : vstep ?X ?Y |- _ ]      =>
        introduce_eq_in s X H; introduce_eq_in s' Y H
    | [ H : vmultistep ?X ?Y |- _ ] =>
        introduce_eq_in s X H; introduce_eq_in s' Y H
    | _ => idtac "No step hypothesis found"
  end.

(**
   Same as [introduce_step_states], but adapted to [pstep] and [pmultistep].
*)
Ltac introduce_step_states_p H vmu s s' :=
  match goal with
    | [ H : pstep ?S ?X ?Y |- _ ]      =>
        introduce_eq_in vmu S H;
        introduce_eq_in s X H;
        introduce_eq_in s' Y H
    | [ H : pmultistep ?S ?X ?Y |- _ ] =>
        introduce_eq_in vmu S H;
        introduce_eq_in s X H;
        introduce_eq_in s' Y H
    | _ => idtac "No step hypothesis found"
  end.

(**
   New tactic notation: [introduce_step_states H s s'] can
   be written [introduce states s s' in H].
*)
Tactic Notation "introduce" "states" ident(s) ident(s') "in" hyp(H) :=
  introduce_step_states H s s'.

(**
   New tactic notation: [introduce_step_states_p H vmu s s'] can
   be written [introduce states vmu s s' in H].
*)
Tactic Notation "introduce" "states" ident(vmu) ident(s) ident(s') "in" hyp(H) :=
  introduce_step_states_p H vmu s s'.

(**
   Splits up a goal [multistep s1 s2] to [step s1 Y]
   and [step Y s2], and similarly for [vmultistep] and [pmultistep].
   This tactic just makes it unnecessary
   to copy-paste s1 and s2 when applying the
   [multiStep]/[vmultistep]/[pmultistep] theorem.
*)
```

```
Ltac split_step_in_goal Y :=
  match goal with
    | |-  multistep ?X ?Z    => apply ( multiStep X Y Z)
    | |- vmultistep ?X ?Z    => apply (vmultiStep X Y Z)
    | |- pmultistep ?S ?X ?Z => apply (pmultiStep S X Y Z)
    | _                      => fail
  end.

(**
   New tactic notation: [split_step_in_goal X] can we written [step X].
*)
Tactic Notation "step" constr(X) := split_step_in_goal X.

(**
   Builds up a (finite) set of identifiers which
   are assigned to in the given command
*)
Fixpoint targets (c : command) : IdSet.t :=
  match c with
    | cAssert _       => IdSet.empty
    | cAssume _       => IdSet.empty
    | cAssign x _     => IdSet.singleton x
    | cSequence c1 c2 => IdSet.union (targets c1) (targets c2)
    | cChoice c1 c2   => IdSet.union (targets c1) (targets c2)
    | cSkip           => IdSet.empty
  end.

(**
   When needing to prove [multistep (ip (cChoice c1 c2) mu)],
   takes the left path and thus changes the goal to [multistep (ip c1 mu)].
   Similarly for [vmultistep] and [pmultistep].
*)
Ltac step_choice_left :=
  match goal with
  | |- multistep  ( ip ( cChoice ?C1 ?C2) ?MU ) ?S =>
        step ( ip C1  MU); [ constructor | idtac ]
  | |- vmultistep (vip (vcChoice ?C1 ?C2) ?VMU) ?S =>
        step (vip C1 VMU); [ constructor | idtac ]
  | |- pmultistep ?VMU (pip (pcChoice ?C1 ?C2)) ?S =>
        step (pip C1); [ constructor | idtac ]
  end.

(**
   When needing to prove [multistep (ip (cChoice c1 c2) mu)],
   takes the right path and thus changes the goal to [multistep (ip c2 mu)].
   Similarly for [vmultistep] and [pmultistep].
*)
Ltac step_choice_right := match goal with
  | |- multistep  ( ip ( cChoice ?C1 ?C2) ?MU ) ?S =>
       step ( ip C2  MU); [ constructor | idtac ]
  | |- vmultistep (vip (vcChoice ?C1 ?C2) ?VMU) ?S =>
       step (vip C2 VMU); [ constructor | idtac ]
  | |- pmultistep ?VMU (pip (pcChoice ?C1 ?C2)) ?S =>
       step (pip C2); [ constructor | idtac ]
  end.
```

```
(**
   New tactic notation: [step_choice_left] can be written [step choice left].
*)
Tactic Notation "step" "choice" "left" := step_choice_left.

(**
   New tactic notation: [step_choice_right] can be written [step choice right].
*)
Tactic Notation "step" "choice" "right" := step_choice_right.

(**
   A few theorems about the steps and multisteps relations.
*)
Lemma multistep_ip_ip :
  forall s c' mu',
    multistep s (ip c' mu') -> exists c, exists mu, s = ip c mu.
Proof. (* 5 lines *) Qed.

Lemma vmultistep_vip_vip :
  forall s c' vmu',
    vmultistep s (vip c' vmu') -> exists c, exists vmu, s = vip c vmu.
Proof. (* 5 lines *) Qed.

Lemma pmultistep_pip_pip :
  forall s c' vmu,
    pmultistep vmu s (pip c') -> exists c, s = pip c.
Proof. (* 5 lines *) Qed.

(**
   If a sequence c1; c2 goes to skip, both c1 and c2 go to skip separately.
*)
Theorem multistep_seq_skip :
  forall c1 c2 mu mu",
    multistep (ip (cSequence c1 c2) mu) (ip cSkip mu") ->
    exists mu', multistep (ip c1 mu) (ip cSkip mu') /\
                multistep (ip c2 mu') (ip cSkip mu").
Proof. (* 18 lines *) Qed.

(**
   If a sequence c1; c2 goes to skip, both c1 and c2 go to skip separately.
*)
Theorem vmultistep_seq_skip :
  forall c1 c2 vmu vmu",
    vmultistep (vip (vcSequence c1 c2) vmu) (vip vcSkip vmu") ->
    exists vmu', vmultistep (vip c1 vmu) (vip vcSkip vmu') /\
                 vmultistep (vip c2 vmu') (vip vcSkip vmu").
Proof. (* 18 lines *) Qed.

(**
   A lemma to turn a proof of [multistep s1 s2] into one of [step s1 s2].
*)
Lemma multistep_step_to_multi :
  forall s1 s2, step s1 s2 -> multistep s1 s2.
Proof. (* 2 lines *) Qed.
```

```
(**
   A lemma to turn a proof of [vmultistep s1 s2] into one of [vstep s1 s2].
*)
Lemma vmultistep_step_to_multi :
  forall s1 s2, vstep s1 s2 -> vmultistep s1 s2.
Proof. (* 2 lines *) Qed.

(**
   A lemma to turn a proof of [pmultistep vmu s1 s2] into one of [pstep vmu s1
s2].
*)
Lemma pmultistep_step_to_multi :
  forall vmu s1 s2, pstep vmu s1 s2 -> pmultistep vmu s1 s2.
Proof. (* 2 lines *) Qed.

(**
   If a sequence c1; c2 fails, either c1 fails, or c1 skips and c2 fails.
*)
Theorem multistep_seq_fail :
  forall c1 c2 mu mu”,
    multistep (ip (cSequence c1 c2) mu) (failure mu”) ->
    multistep (ip c1 mu) (failure mu”) \/
    (exists mu’, multistep (ip c1 mu) (ip cSkip mu’) /\ multistep (ip c2 mu’)
(failure mu”)).
Proof. (* 26 lines *) Qed.

(**
   If a sequence c1; c2 fails, either c1 fails, or c1 skips and c2 fails.
*)
Theorem vmultistep_seq_fail :
  forall c1 c2 vmu vmu”,
    vmultistep (vip (vcSequence c1 c2) vmu) (vfailure vmu”) ->
    vmultistep (vip c1 vmu) (vfailure vmu”) \/
    (exists vmu’, vmultistep (vip c1 vmu) (vip vcSkip vmu’) /\
                  vmultistep (vip c2 vmu’) (vfailure vmu”)).
Proof. (* 26 lines *) Qed.

(**
   If a choice c1 [] c2 skips, then c1 skips or c2 skips.
*)
Theorem multistep_choice_skip :
  forall c1 c2 mu mu’,
    multistep (ip (cChoice c1 c2) mu) (ip cSkip mu’) ->
    multistep (ip c1 mu) (ip cSkip mu’) \/ multistep (ip c2 mu) (ip cSkip mu’).
Proof. (* 3 lines *) Qed.

(**
   If a choice c1 [] c2 fails, then either c1 of c2 fails.
*)
Theorem multistep_choice_fail :
  forall c1 c2 mu mu’,
    multistep (ip (cChoice c1 c2) mu) (failure mu’) ->
    multistep (ip c1 mu) (failure mu’) \/ multistep (ip c2 mu) (failure mu’).
Proof. (* 3 lines *) Qed.
```

```
(**
   If a choice c1 [] c2 skips, then c1 skips or c2 skips.
*)
Theorem vmultistep_choice_skip :
  forall c1 c2 vmu vmu',
    vmultistep (vip (vcChoice c1 c2) vmu) (vip vcSkip vmu') ->
    vmultistep (vip c1 vmu) (vip vcSkip vmu') \/
    vmultistep (vip c2 vmu) (vip vcSkip vmu').
Proof. (* 3 lines *) Qed.

(**
   If a choice c1 [] c2 fails, then either c1 of c2 fails.
*)
Theorem vmultistep_choice_fail :
  forall c1 c2 vmu vmu',
    vmultistep (vip (vcChoice c1 c2) vmu) (vfailure vmu') ->
    vmultistep (vip c1 vmu) (vfailure vmu') \/
    vmultistep (vip c2 vmu) (vfailure vmu').
Proof. (* 3 lines *) Qed.

(**
   Transitivity of multistep
*)
Theorem multistep_trans :
  forall s1 s2 s3, multistep s1 s2 -> multistep s2 s3 -> multistep s1 s3.
Proof. (* 9 lines *) Qed.

(**
   Transitivity of vmultistep
*)
Theorem vmultistep_trans :
  forall s1 s2 s3, vmultistep s1 s2 -> vmultistep s2 s3 -> vmultistep s1 s3.
Proof. (* 9 lines *) Qed.

(**
   Transitivity of pmultistep
*)
Theorem pmultistep_trans : forall vmu s1 s2 s3,
  pmultistep vmu s1 s2 -> pmultistep vmu s2 s3 -> pmultistep vmu s1 s3.
Proof. (* 9 lines *) Qed.

(**
   [command]s are at least 1 big.
*)
Lemma command_metric_min_size :
  forall c, 1 <= command_metric c.
Proof. (* 1 lines *) Qed.

Hint Resolve command_metric_min_size.

(**
   [vcommand]s are at least 1 big.
*)
Lemma vcommand_metric_min_size :
  forall c, 1 <= vcommand_metric c.
Proof. (* 1 lines *) Qed.
```

```
Hint Resolve vcommand_metric_min_size.

(**
   [pcommand]s are at least 1 big.
*)
Lemma pcommand_metric_min_size :
  forall c, 1 <= pcommand_metric c.
Proof. (* 1 lines *) Qed.

Hint Resolve pcommand_metric_min_size.

(**
   Shows that the [step] relation reduces the command's size, i.e. evaluation
   of a program is guaranteed to end.
*)
Theorem step_monotonic_commands :
  forall c c' mu mu',
    step (ip c mu) (ip c' mu') -> command_metric c' < command_metric c.
Proof. (* 28 lines *) Qed.

(**
   Shows that the [vstep] relation reduces the command's size, i.e. evaluation
   of a program is guaranteed to end.
*)
Theorem vstep_monotonic_commands :
  forall c c' vmu vmu',
    vstep (vip c vmu) (vip c' vmu') -> vcommand_metric c' < vcommand_metric c.
Proof. (* 28 lines *) Qed.

(**
   Shows that the [pstep] relation reduces the command's size, i.e. evaluation
   of a program is guaranteed to end.
*)
Theorem pstep_monotonic_commands :
  forall c c' vmu,
    pstep vmu (pip c) (pip c') -> pcommand_metric c' < pcommand_metric c.
Proof. (* 24 lines *) Qed.

Theorem step_monotonic_states :
  forall s s',
    step s s' -> state_metric s' < state_metric s.
Proof. (* 6 lines *) Qed.

Theorem vstep_monotonic_states :
  forall s s',
    vstep s s' -> vstate_metric s' < vstate_metric s.
Proof. (* 6 lines *) Qed.

Theorem pstep_monotonic_states :
  forall s s' vmu,
    pstep vmu s s' -> pstate_metric s' < pstate_metric s.
Proof. (* 6 lines *) Qed.

Theorem multistep_monotonic_commands :
  forall c c' mu mu',
    multistep (ip c mu) (ip c' mu') -> command_metric c' <= command_metric c.
Proof. (* 13 lines *) Qed.
```

```
Theorem vmultistep_monotonic_commands :
  forall c c' vmu vmu',
    vmultistep (vip c vmu) (vip c' vmu') ->
    vcommand_metric c' <= vcommand_metric c.
Proof. (* 13 lines *) Qed.

Theorem pmultistep_monotonic_commands :
  forall c c' vmu,
    pmultistep vmu (pip c) (pip c') -> pcommand_metric c' <= pcommand_metric c.
Proof. (* 12 lines *) Qed.

Theorem multistep_monotonic_states :
  forall s s',
    multistep s s' -> state_metric s' <= state_metric s.
Proof. (* 7 lines *) Qed.

Theorem vmultistep_monotonic_states :
  forall s s',
    vmultistep s s' -> vstate_metric s' <= vstate_metric s.
Proof. (* 7 lines *) Qed.

Theorem pmultistep_monotonic_states :
  forall s s' vmu,
    pmultistep vmu s s' -> pstate_metric s' <= pstate_metric s.
Proof. (* 7 lines *) Qed.

Theorem step_asymmetric :
  forall s s',
    step s s' -> ~ step s' s.
Proof. (* 4 lines *) Qed.

Theorem vstep_asymmetric :
  forall s s',
    vstep s s' -> ~ vstep s' s.
Proof. (* 4 lines *) Qed.

Theorem pstep_asymmetric :
  forall vmu s s',
    pstep vmu s s' -> ~ pstep vmu s' s.
Proof. (* 4 lines *) Qed.

Theorem multistep_antisymmetric :
  forall s s',
    multistep s s' -> multistep s' s -> s = s'.
Proof. (* 11 lines *) Qed.

Theorem vmultistep_antisymmetric :
  forall s s',
    vmultistep s s' -> vmultistep s' s -> s = s'.
Proof. (* 11 lines *) Qed.

Theorem pmultistep_antisymmetric :
  forall s s' vmu,
    pmultistep vmu s s' -> pmultistep vmu s' s -> s = s'.
Proof. (* 11 lines *) Qed.
```

```
(**
   Converts an [id] to a [vid], using a version map [v].
*)
Definition id_to_vid (id : id) (v : vmap) : vid := (id, v id).

(**
   Converts a [vstore] to a [store], using a version map [v].
*)
Definition vstore_to_store (vmu : vstore) (v : vmap) : store :=
  fun id => vmu (id_to_vid id v).

(**
   Converts an [expr] to a [vexpr], using a version map [v].
*)
Definition expr_to_vexpr (e : expr) (v : vmap) : vexpr :=
  fun vmu => e (vstore_to_store vmu v).

(**
   Converts a [command] to a [vcommand], using a version map [v].
*)
Fixpoint command_to_vcommand (c : command) (v : vmap) : vcommand :=
  match c with
    | cAssert e       => vcAssert (expr_to_vexpr e v)
    | cAssume e       => vcAssume (expr_to_vexpr e v)
    | cAssign x e     => vcAssign (id_to_vid x v) (expr_to_vexpr e v)
    | cSequence c1 c2 => vcSequence (command_to_vcommand c1 v)
                                    (command_to_vcommand c2 v)
    | cSkip           => vcSkip
    | cChoice c1 c2   => vcChoice (command_to_vcommand c1 v)
                                  (command_to_vcommand c2 v)
  end.

Theorem vstore_to_store_surjective :
  forall v mu,
    exists vmu, mu = vstore_to_store vmu v.
Proof. (* 6 lines *) Qed.

Theorem expr_to_vexpr_injective : forall e e' v,
  expr_to_vexpr e v = expr_to_vexpr e' v -> e = e'.
Proof. (* 11 lines *) Qed.

Theorem command_to_vcommand_injective : forall c c' v,
  command_to_vcommand c v = command_to_vcommand c' v -> c = c'.
Proof. (* 33 lines *) Qed.

(**
   Shows that [step] and [vstep] behave the same when not ending up in failure.
*)
Theorem step_sim_vstep : forall c c' mu mu' vmu v,
  let vc := command_to_vcommand c v in
  let vc' := command_to_vcommand c' v in
    step (ip c mu) (ip c' mu') ->
    store_sync_vstore mu v vmu ->
    exists vmu', vstep (vip vc vmu) (vip vc' vmu') /\ store_sync_vstore mu' v
vmu'.
Proof. (* 92 lines *) Qed.
```

```
(**
   Shows that [step] and [vstep] behave the same when ending up in failure.
*)
Theorem step_sim_vstep_fail : forall c mu mu' vmu v,
  let vc := command_to_vcommand c v in
    step (ip c mu) (failure mu') ->
    store_sync_vstore mu v vmu ->
    exists vmu', vstep (vip vc vmu) (vfailure vmu') /\ store_sync_vstore mu' v
vmu'.
Proof. (* 42 lines *) Qed.

(**
   Shows that [multistep] and [vmultistep] behave similarly
   when not ending up in failure.
*)
Theorem multistep_sim_vmultistep : forall c c' mu mu' vmu v,
  let vc := command_to_vcommand c v in
  let vc' := command_to_vcommand c' v in
    multistep (ip c mu) (ip c' mu') ->
    store_sync_vstore mu v vmu ->
    exists vmu', vmultistep (vip vc vmu) (vip vc' vmu') /\
                 store_sync_vstore mu' v vmu'.
Proof. (* 33 lines *) Qed.

(**
   Shows that [multistep] can't hop from one failure state to
   another one with a different store.
*)
Lemma multistep_fail_fail_equal_stores :
  forall mu mu',
    multistep (failure mu) (failure mu') -> mu = mu'.
Proof. (* 11 lines *) Qed.

(**
   Shows that [vmultistep] can't hop from one failure state
   to another one with a different store.
*)
Lemma vmultistep_fail_fail_equal_stores :
  forall vmu vmu',
    vmultistep (vfailure vmu) (vfailure vmu') -> vmu = vmu'.
Proof. (* 11 lines *) Qed.

(**
   Shows that [multistep] and [vmultistep] behave similarly
   when ending up in failure.
*)
Theorem multistep_sim_vmultistep_fail : forall c mu mu' vmu v,
  let vc := command_to_vcommand c v in
    multistep (ip c mu) (failure mu') ->
    store_sync_vstore mu v vmu ->
    exists vmu', vmultistep (vip vc vmu) (vfailure vmu') /\
                 store_sync_vstore mu' v vmu'.
Proof. (* 43 lines *) Qed.
```

```
(**
   If we start of in a [vcommand] which is the
   result of the translation of a [command],
   the [step] relation will lead us only to
   [vcommand]s which themselves are also
   translations of some [command].

   In other words, the [command_to_vcommand] function
   is invertible, with some strings attached.
*)
Lemma vcommand_to_command :
  forall c vc' vmu vmu' v,
    vstep (vip (command_to_vcommand c v) vmu) (vip vc' vmu') ->
    exists c', vc' = command_to_vcommand c' v.
Proof. (* 59 lines *) Qed.

(**
   [vstep] and [step] behave similarly when not leading to a failure state.
*)
Theorem vstep_sim_step : forall v c c' mu vmu vmu',
  let vc := command_to_vcommand c v
  in
  let vc' := command_to_vcommand c' v
  in
    store_sync_vstore mu v vmu ->
    vstep (vip vc vmu) (vip vc' vmu') ->
    exists mu', step (ip c mu) (ip c' mu') /\
                store_sync_vstore mu' v vmu'.
Proof. (* 119 lines *) Qed.

(**
   [vstep] and [step] behave similarly when leading to a failure state.
*)
Theorem vstep_sim_step_fail : forall v c mu vmu vmu',
  let vc := command_to_vcommand c v
  in
    store_sync_vstore mu v vmu ->
    vstep (vip vc vmu) (vfailure vmu') ->
    exists mu', step (ip c mu) (failure mu') /\
                store_sync_vstore mu' v vmu'.
Proof. (* 41 lines *) Qed.

(**
   [vmultistep] and [multistep] behave similarly when not ending in failure.
*)
Theorem vmultistep_sim_multistep : forall c c' mu vmu vmu' v,
  let vc := command_to_vcommand c v in
  let vc' := command_to_vcommand c' v in
    store_sync_vstore mu v vmu ->
    vmultistep (vip vc vmu) (vip vc' vmu') ->
    exists mu', multistep (ip c mu) (ip c' mu') /\
                store_sync_vstore mu' v vmu'.
Proof. (* 40 lines *) Qed.
```

```
(**
   [vmultistep] and [multistep] behave similarly when ending in failure.
*)
Theorem vmultistep_sim_multistep_fail : forall c mu vmu vmu' v,
  let vc := command_to_vcommand c v in
    store_sync_vstore mu v vmu ->
    vmultistep (vip vc vmu) (vfailure vmu') ->
    exists mu', multistep (ip c mu) (failure mu') /\
                store_sync_vstore mu' v vmu'.
Proof. (* 50 lines *) Qed.

(**
   Our goal is to verify that [vstep] and [pstep] behave similarly.
   In order to do this, we need to translate [vcommand]s to [pcommand]s
   (this translation must not be confused with the passification step later on).
   The problem is that [vcommand]s can contain assignments, while
   [pcommand]s cannot. Thus, we can only compare assignment-less programs.
   This function determines whether a [vcommand] contains no assignments.
*)
Fixpoint no_assignments (c : vcommand) : Prop :=
  match c with
    | vcAssert _       => True
    | vcAssume _       => True
    | vcAssign _ _     => False
    | vcSequence c1 c2 => no_assignments c1 /\ no_assignments c2
    | vcSkip           => True
    | vcChoice c1 c2   => no_assignments c1 /\ no_assignments c2
  end.

(**
   Using [vstep], if a program does not contain assignments,
   the store remains unchanged.
*)
Theorem no_assignments_preserves_store :
  forall c c' mu mu',
    no_assignments c -> vstep (vip c mu) (vip c' mu') -> mu = mu'.
Proof. (* 8 lines *) Qed.

(**
   Using [vstep], if a program does not contain assignments,
   the store remains unchanged.
*)
Theorem no_assignments_preserves_store_fail :
  forall c mu mu',
    no_assignments c -> vstep (vip c mu) (vfailure mu') -> mu = mu'.
Proof. (* 7 lines *) Qed.

(**
   [vstep] reduces commands without assignments to commands
   which also do not contain assignments. In other words,
   the [vstep] reduction rules do not introduce assignemnts.
*)
Theorem vstep_preserves_no_assignments : forall c c' mu mu',
  no_assignments c -> vstep (vip c mu) (vip c' mu') -> no_assignments c'.
Proof. (* 44 lines *) Qed.
```

```
(**
   Transform a [vcommand] into an equivalent [pcommand],
   on the condition that the [vcommand]
   does not contain assignments.
*)
Definition vcommand_to_pcommand (vc : vcommand) (H : no_assignments vc) :
pcommand.
  induction vc; intros.
  exact (pcAssert v).
  exact (pcAssume v).
  simpl in H.
  contradiction.
  simpl in H.
  destruct H.
  exact (pcSequence (IHvc1 H) (IHvc2 H0)).
  exact pcSkip.
  simpl in H; destruct H.
  exact (pcChoice (IHvc1 H) (IHvc2 H0)).
Defined.

(**
   [vstep] behaves similarly to [pstep] for assignmentless programs.
*)
Theorem vstep_sim_pstep :
  forall vc vc' H vmu vmu' (H0 : vstep (vip vc vmu) (vip vc' vmu')),
    let pc := vcommand_to_pcommand vc H
    in
      let pc' :=
        vcommand_to_pcommand vc'
                             (vstep_preserves_no_assignments _ _ _ _ H H0)
      in
        pstep vmu (pip pc) (pip pc').
Proof. (* 62 lines *) Qed.

(**
   [vstep] behaves similarly to [pstep] for assignmentless programs.
*)
Theorem vstep_sim_pstep_fail :
  forall vc H vmu vmu' (H0 : vstep (vip vc vmu) (vfailure vmu')),
    let pc := vcommand_to_pcommand vc H
    in
      pstep vmu (pip pc) pfailure.
Proof. (* 22 lines *) Qed.

(**
   If one starts with an assignmentless command,
   one will always end up in an assignmentless command using [vmultistep].
*)
Theorem vmultistep_preserves_no_assignments :
  forall c c' vmu vmu',
    no_assignments c ->
    vmultistep (vip c vmu) (vip c' vmu') ->
    no_assignments c'.
Proof. (* 20 lines *) Qed.
```

```
(**
   Translates a [pcommand] to an equivalent [vcommand].
*)
Fixpoint pcommand_to_vcommand (pc : pcommand) : vcommand :=
  match pc with
    | pcAssert e       => vcAssert e
    | pcAssume e       => vcAssume e
    | pcSkip           => vcSkip
    | pcSequence c1 c2 =>
      vcSequence (pcommand_to_vcommand c1) (pcommand_to_vcommand c2)
    | pcChoice c1 c2   =>
      vcChoice (pcommand_to_vcommand c1) (pcommand_to_vcommand c2)
  end.

(**
   Shows that [pstep] behaves similarly to [vstep].
*)
Theorem pstep_sim_vstep :
  forall pc pc' vmu,
    let vc := pcommand_to_vcommand pc in
      let vc' := pcommand_to_vcommand pc' in
        pstep vmu (pip pc) (pip pc') -> vstep (vip vc vmu) (vip vc' vmu).
Proof. (* 35 lines *) Qed.

(**
   Shows that [pstep] behaves similarly to [vstep].
*)
Theorem pstep_sim_vstep_fail :
  forall pc vmu,
    let vc := pcommand_to_vcommand pc in
      pstep vmu (pip pc) pfailure -> vstep (vip vc vmu) (vfailure vmu).
Proof. (* 15 lines *) Qed.

(**
   Shows that [pmultistep] behaves similarly to [vmultistep].
*)
Theorem pmultistep_sim_vmultistep :
  forall pc pc' vmu,
    let vc := pcommand_to_vcommand pc
    in
      let vc' := pcommand_to_vcommand pc'
      in
        pmultistep vmu (pip pc) (pip pc') ->
        vmultistep (vip vc vmu) (vip vc' vmu).
Proof. (* 24 lines *) Qed.

(**
   Shows that [pmultistep] behaves similarly to [vmultistep].
*)
Theorem pmultistep_sim_vmultistep_fail :
  forall pc vmu,
    let vc := pcommand_to_vcommand pc in
      pmultistep vmu (pip pc) pfailure ->
      vmultistep (vip vc vmu) (vfailure vmu).
Proof. (* 22 lines *) Qed.
```

```
(**
   We wish to show that execution only gets stuck on
   a) a "failing" assume,
   b) a failure state,
   c) skip.
   Of these, only the former is hard (relatively to b and c)
   to recognize, as the assume may be hidden deep in a tree structure.

   [nested_assume] describes commands where execution will
   need to deal with an assume-command first.
   [nested_assume e c] states that [c] is a command
   where [cAssert e] is the next command to be executed.
*)
Inductive nested_assume (e : expr) : command -> Prop :=
| naAssume   : nested_assume e (cAssume e)
| naSequence : forall c1 c2,
                 nested_assume e c1 -> nested_assume e (cSequence c1 c2).

(**
   Defines the general form of a stuck state. Our intention
   is to prove that no reduction rule applies on a command for
   which [stuck_state] is true, and conversely, that
   if a state [s] cannot be reduced any further, [stuck_state] must be true.
*)
Inductive stuck_state : state -> Prop :=
| ssAssume : forall c (e : expr) mu,
               e mu <> T -> nested_assume e c -> stuck_state (ip c mu)
| ssSkip   : forall mu, stuck_state (ip cSkip mu)
| ssFail   : forall mu, stuck_state (failure mu).

(**
   A state [s] is [decidable_reducible] if either there exists a state [s']
   for which    [step s s'] is true (i.e. it is reducible), or,
   [s] is a stuck state.
*)
Definition decidable_reducible (s : state) :=
  { s' : state | step s s' } + { stuck_state s }.

(**
   States that we can determine in finite time whether
   a command [c] is the skip command or not.
*)
Lemma decidable_skip : forall c, {c = cSkip} + {c <> cSkip}.
Proof. (* 2 lines *) Qed.

(**
   States that every state [s] is [decidable_reducible], i.e.
   that either it there exists a state [s'] for which [state s s'] is
   true, or that [stuck_state s] holds.
*)
Theorem decidable_reducible_states : forall s, decidable_reducible s.
Proof. (* 46 lines *) Qed.
```

```
(**
   We show the soundness of [stuck_state], i.e.
   that if [stuck_state] claims a state s is false,
   that there indeed is no applicable reduction step.
*)
Theorem soundness_stuck_state :
  forall s, stuck_state s ->
    ~ exists s', step s s'.
Proof. (* 57 lines *) Qed.

(**
   We show the completeness of [stuck_state]:
   if there is no reduction step starting from s,
   s is indeed recognized by [stuck_state] as being stuck.
*)
Theorem completeness_stuck_state :
  forall s, (~ exists s', step s s') -> stuck_state s.
Proof. (* 7 lines *) Qed.

(**
   [apply_multistep_transitivity T] splits a goal [multistep S S'] into
   two subgoals [multistep S T] and [multistep T S'].
   Works similarly for [vmultistep] and [pmultistep].

«
  s : state
  s' : state
  s" : state
  ============================
   multistep s s"

apply_multistep_transitivity s'.

subgoal 1 is:

  s : state
  s' : state
  s" : state
  ============================
   multistep s s'

subgoal 2 is:

  s : state
  s' : state
  s" : state
  ============================
   multistep s' s"
» *)
Ltac apply_multistep_transitivity t :=
  match goal with
  | |- multistep ?X ?Y =>
       first [ apply (multistep_trans X t Y)
             | idtac "Failed to apply transitivity" ]
  | |- vmultistep ?X ?Y =>
       first [ apply (vmultistep_trans X t Y)
             | idtac "Failed to apply transitivity" ]
  | |- pmultistep ?S ?X ?Y =>
      first [ apply (pmultistep_trans S X t Y)
            | idtac "Failed to apply transitivity" ]
  end.

(**
   New tactic notation: [apply_multistep_transitivity t] can
   be written as [step transitivity with t].
*)
Tactic Notation "step" "transitivity" "with" constr(t) :=
  apply_multistep_transitivity t.

(**
   Lifts [c1 --*> c1'] into a sequence: [c1; c2 --*> c1'; c2]
*)
Theorem multistep_lift_seq :
  forall c c' c2 mu mu',
    multistep (ip c mu) (ip c' mu') ->
    multistep (ip (cSequence c c2) mu) (ip (cSequence c' c2) mu').
Proof. (* 12 lines *) Qed.

(**
   Lifts [c1 --*> c1'] into a sequence: [c1; c2 --*> c1'; c2]
*)
Theorem vmultistep_lift_seq :
  forall c c' c2 vmu vmu',
    vmultistep (vip c vmu) (vip c' vmu') ->
    vmultistep (vip (vcSequence c c2) vmu) (vip (vcSequence c' c2) vmu').
Proof. (* 12 lines *) Qed.

(**
   Lifts [c1 --*> c1'] into a sequence: [c1; c2 --*> c1'; c2]
*)
Theorem pmultistep_lift_seq :
  forall c c' c2 vmu,
    pmultistep vmu (pip c) (pip c') ->
    pmultistep vmu (pip (pcSequence c c2)) (pip (pcSequence c' c2)).
Proof. (* 17 lines *) Qed.

(**
   Lifts [c --*> fail] into a sequence: [c; c' --*> fail]
*)
Lemma multistep_lift_seq_fail :
  forall c c' mu mu',
    multistep (ip c mu) (failure mu') ->
    multistep (ip (cSequence c c') mu) (failure mu').
Proof. (* 18 lines *) Qed.

(**
   Lifts [c --*> fail] into a sequence: [c; c' --*> fail]
*)
Lemma vmultistep_lift_seq_fail :
  forall c c' vmu vmu',
    vmultistep (vip c vmu) (vfailure vmu') ->
    vmultistep (vip (vcSequence c c') vmu) (vfailure vmu').
Proof. (* 18 lines *) Qed.
```

```
(**
   Lifts [c --*> fail] into a sequence: [c; c' --*> fail]
*)
Lemma pmultistep_lift_seq_fail :
  forall c1 c2 vmu,
    pmultistep vmu (pip c1) pfailure ->
    pmultistep vmu (pip (pcSequence c1 c2)) pfailure.
Proof. (* 16 lines *) Qed.

(**
   ** Single Assignment
*)

(**
   [delta_id A f g d] expresses that forall all [id]s [x], [f x = g x], except
   possibly for the [id]s contained in the set [d], where [f] and [g]
   are total functions with domain [id] and range [A].
*)
Definition delta_id (A : Set) (f g : id -> A) (d : IdSet.t) :=
  forall x : id, f x = g x \/ IdSet.In x d.

(**
   A few theorems about delta_id.
*)

Lemma delta_id_x_x :
  forall A x d, delta_id A x x d.
Proof. (* 2 lines *) Qed.

Hint Resolve delta_id_x_x.

Lemma delta_id_unionl :
  forall A f g d1 d2,
    delta_id A f g d1 -> delta_id A f g (IdSet.union d1 d2).
Proof. (* 5 lines *) Qed.

Lemma delta_id_unionr :
  forall A f g d1 d2,
    delta_id A f g d2 -> delta_id A f g (IdSet.union d1 d2).
Proof. (* 6 lines *) Qed.

Lemma delta_id_combine :
  forall A f g h d1 d2,
    delta_id A f g d1 ->
    delta_id A g h d2 ->
    delta_id A f h (IdSet.union d1 d2).
Proof. (* 3 lines *) Qed.

Lemma delta_id_extend :
  forall A f g d1 d2,
    delta_id A f g d1 -> IdSet.Subset d1 d2 -> delta_id A f g d2.
Proof. (* 4 lines *) Qed.

(**
   We specialize [delta_id] for [store]s.
*)
Definition store_delta := delta_id value.
```

```
(**
   We specialize delta_id for [vmap]s.
*)
Definition vmap_delta := delta_id nat.

Definition vmap_delta_combine := delta_id_combine nat.

Theorem store_delta_mu_mu :
  forall (mu : store) (d : IdSet.t),
    store_delta mu mu d.
Proof. (* 2 lines *) Qed.

Theorem vmap_delta_v_v :
  forall (v : vmap) (d : IdSet.t),
    vmap_delta v v d.
Proof. (* 2 lines *) Qed.

Theorem vmap_delta_extend : forall v1 v2 d1 d2,
    vmap_delta v1 v2 d1 -> IdSet.Subset d1 d2 -> vmap_delta v1 v2 d2.
Proof. (* 2 lines *) Qed.

Hint Resolve store_delta_mu_mu.
Hint Resolve vmap_delta_v_v.

Definition store_delta_unionl := delta_id_unionl value.
Definition vmap_delta_unionl := delta_id_unionl nat.
Definition store_delta_unionr := delta_id_unionr value.
Definition vmap_delta_unionr := delta_id_unionr nat.

(**
   We prove that the [step] relation keeps the store unchanged, except for
   the command's targets (the set of identifiers appearing on the left side
   of assignments).
*)
Theorem step_store_delta : forall c c' mu mu' d,
  IdSet.Subset (targets c) d ->
  step (ip c mu) (ip c' mu') ->
  store_delta mu mu' d.
Proof. (* 22 lines *) Qed.

(**
   The targets set of a command will not grow through the [step] relation.
*)
Theorem step_targets_subset : forall c c' mu mu',
  step (ip c mu) (ip c' mu') -> IdSet.Subset (targets c') (targets c).
Proof. (* 12 lines *) Qed.

(**
   Given the [store]s [mu], [mu'] and [mu"], and
   [mu x = mu' x] for all [x] except for those in some set [d],
   and [mu' x = mu" x] for all [x] except for those in that same set [d],
   then [mu x = mu" x] for all [x] except for those in [d].
*)
Theorem store_delta_trans :
  forall mu mu' mu" d,
    store_delta mu mu' d -> store_delta mu' mu" d -> store_delta mu mu" d.
Proof. (* 3 lines *) Qed.
```

```
(**
   step_store_delta adapted to the multistep relation.
*)
Theorem multistep_store_delta : forall c c” mu mu” d,
    IdSet.Subset (targets c) d ->
    multistep (ip c mu) (ip c” mu”) ->
    store_delta mu mu” d.
Proof. (* 14 lines *) Qed.

(**
   Checks if two versioned stores under a certain vmap are equivalent.
*)
Definition vstore_sync_vstore (mu : vstore) (v v’ : vmap) (mu’ : vstore) :=
  forall x : id, mu (x, v x) = mu’ (x, v’ x).

(**
   Generates a command which copies across versions:
«
   copy_vcmd x n m   ===   x_m := x_n
»*)
Definition copy_vcmd (x : id) (n m : nat) :=
  vcAssign (x, m) (fun (vmu : vstore) => vmu (x, n)).

(**
   Shows that [copy_vcmd_works] as expected: execution of [copy_vcmd x n m]
   in a store [vmu] will lead to a new store [vmu’]
   where [vmu’ (x, m) = vmu (x, n)] (i.e. [x_m] is now equal to [x_n]),
   while all other store bindings remain unchanged.
*)
Theorem copy_vcmd_works :
  forall x n m vmu, exists vmu’,
    vstep (vip (copy_vcmd x n m) vmu) (vip vcSkip vmu’) /\
    vmu’ (x, m) = vmu (x, n) /\
    forall y k, (x <> y \/ k <> m) -> vmu (y, k) = vmu’ (y, k).
Proof. (* 17 lines *) Qed.

(**
   Right fold on lists of [id]s.
*)
Fixpoint foldr (B : Set) (f : id -> B -> B) (s : list id) : B -> B :=
  fun i => match s with
             | nil    => i
             | x :: l => f x (foldr B f l i)
           end.

(**
   Right fold on finite [id]-sets
*)
Definition fset_foldr (B : Set) (f : id -> B -> B) (s : IdSet.t) (init : B) : B.
  intros.
  unfold IdSet.t in s.
  destruct s.
  unfold IdSet.Raw.t in this.
  apply (foldr B f this init).
Defined.
```

```
(**
   If the [id] [x] is in need of a synchronisation (i.e. the version
   map [v] and [v'] assign differing versions to [x]) this function
   generates an assignment command. c is a "continuation" so as to make
   it possible to create a chain of multiple assignments.

   Thus, if [v x = v' x], then [insert_copy_vcmd v v' x c] doesn't need to
   produce a synchronizing assignment command and just returns [c].
   Conversely, if [v x <> v' x], the sequence [copy_vcmd x (v x) (v' x); c]
   will be returned.
*)
Definition insert_copy_vcmd v v' x c :=
  if decidable_eq_id (v x) (v' x)
  then c
  else (vcSequence (copy_vcmd x (v x) (v' x)) c).

(**
   Given two version maps v and v' and a set of identifiers,
   sync_vcommand generates a command which
   perform a "store synchronization" from v to v'.

   Example:
«
   mu = { x_4 -> 8, y_3 -> 14, z_5 -> 2, ... }
   v  = { x -> 4, y -> 3, z -> 5, ... }
   v' = { x -> 7, y -> 4, z -> 5, ... }
»
   x and y are assigned different versions by v and v' (4 vs 7, 3 vs 4),
   meaning they need synchronization. sync_vcommand thus generates
«
   x_7 := x_4;
   y_4 := y_3;
   skip
»*)
Definition sync_vcommand (ids : IdSet.t) (v v' : vmap) : vcommand :=
  (fset_foldr vcommand (insert_copy_vcmd v v') ids vcSkip).

(**
   Shows that synchronization commands evaluate to skip.
*)
Theorem sync_vcommand_goes_to_skip : forall ids v v' vmu, exists vmu',
    vmultistep (vip (sync_vcommand ids v v') vmu) (vip vcSkip vmu').
Proof. (* 32 lines *) Qed.

(**
   Shows execution of a synchronization commands always
   ends up in the same state.
*)
Theorem sync_vcommand_determinism :
  forall ids v v' vmu vmu1' vmu2',
    let c := sync_vcommand ids v v' in
      vmultistep (vip c vmu) (vip vcSkip vmu1') ->
      vmultistep (vip c vmu) (vip vcSkip vmu2') ->
      vmu1' = vmu2'.
Proof. (* 53 lines *) Qed.
```

```
Lemma sync_vcommand_empty_delta :
  forall ids v v' vmu vmu',
    equivalent_functions v v' ->
    vmultistep (vip (sync_vcommand ids v v') vmu) (vip vcSkip vmu') ->
    vstore_sync_vstore vmu v v' vmu'.
Proof. (* 26 lines *) Qed.

(**
   A localized version of [vstore_sync_vstore]:
   it need only apply for the [id]s contained in [D].
*)
Definition vstore_sync_vstore_local (mu : vstore)
                                    (v v' : vmap)
                                    (mu' : vstore)
                                    (D : IdSet.t) :=
  forall x : id, IdSet.In x D -> mu (x, v x) = mu' (x, v' x).

(**
   Store bindings for all versions of
   identifiers not in [D] remain untouched.
*)
Lemma sync_vcommand_preservation :
  forall D a v v' vmu vmu' n,
    ~ IdSet.In a D ->
    vmultistep (vip (sync_vcommand D v v') vmu)
               (vip vcSkip vmu') ->
    vmu (a, n) = vmu' (a, n).
Proof. (* 54 lines *) Qed.

Opaque decidable_eq_vid.

(**
   Auxiliary for sync_vcommand_works.
*)
Lemma sync_vcommand_works_aux :
  forall D v v' vmu vmu',
    vmultistep (vip (sync_vcommand D v v') vmu)
               (vip vcSkip vmu') ->
    vstore_sync_vstore_local vmu v v' vmu' D.
Proof. (* 88 lines *) Qed.

(**
   Given two [vmap]s [v] and [v'] whose mappings are the same except
   for the identifiers contained in [ids], then stepping through
   [sync_vcommand] starting with store [mu] results
   in a new store [mu'] which is synchronized with [mu] with
   respect to [v] and [v'].
*)
Theorem sync_vcommand_works :
  forall ids v v' vmu vmu',
    vmap_delta v v' ids ->
    vmultistep (vip (sync_vcommand ids v v') vmu)
               (vip vcSkip vmu') ->
    vstore_sync_vstore vmu v v' vmu'.
Proof. (* 17 lines *) Qed.
```

```
(**
   A kind of transitivity.
 *)
Theorem combine_vmaps :
  forall mu vmu vmu' v v',
    store_sync_vstore mu v vmu ->
    vstore_sync_vstore vmu v v' vmu' ->
    store_sync_vstore mu v' vmu'.
Proof. (* 5 lines *) Qed.

Transparent decidable_eq_vid.

(**
   Joins two [vmap]s together by taking the maximum version of each [id].
*)
Definition join (v1 v2 : vmap) :=
  fun x => max (v1 x) (v2 x).

(**
   Transforms the [command] [c] to an equivalent [vcommand].
*)
Fixpoint transform_sa (c : command) (v : vmap) : (vcommand * vmap) :=
  match c with
    | cAssert e       => (vcAssert (version_expr e v), v)
    | cAssume e       => (vcAssume (version_expr e v), v)
    | cAssign x e     => (vcAssign (x, S (v x)) (version_expr e v), inc v x)
    | cSequence c1 c2 => let (c1', v' ) := transform_sa c1 v in
                         let (c2', v") := transform_sa c2 v' in
                             (vcSequence c1' c2', v")
    | cSkip           => (vcSkip, v)
    | cChoice c1 c2   => let (c1', v1') := transform_sa c1 v in
                         let (c2', v2') := transform_sa c2 v in
                         let t1 := targets c1 in
                         let t2 := targets c2 in
                         let v' := join v1' v2' in
                         let t  := IdSet.union t1 t2 in
                         let d1 := sync_vcommand t v1' v' in
                         let d2 := sync_vcommand t v2' v' in
                           (vcChoice (vcSequence c1' d1) (vcSequence c2' d2), v')
  end.

Theorem sync_vcommand_size : forall D v v',
  let c := sync_vcommand D v v' in
    vcommand_metric c <= 3 * IdSet.cardinal D + 1.
Proof. (* 66 lines *) Qed.

Theorem targets_cardinality_le :
  forall c : command, IdSet.cardinal (targets c) <= command_metric c.
Proof. (* 10 lines *) Qed.

Theorem linear_sync_vcommand :
  forall c v v',
    let vc := sync_vcommand (targets c) v v' in
      vcommand_metric vc <= 3 * command_metric c + 1.
Proof. (* 6 lines *) Qed.
```

```
(**
   We show that the SA transformation is quadratic in size.
*)
Theorem quadratic_sa_transformation : forall c v,
  let (c', _) := transform_sa c v in
  let x := command_metric c in
    vcommand_metric c' <= 5 * x * x + 5 * x.
Proof. (* 165 lines *) Qed.

(**
   Some tests to make sure the algorithms work as expected.
   Of course, this does not count as proof of correctness.
*)
Module Tests.

  Import IdSet.

  (**
     We temporarily break open the id abstraction, so that
     we can define programs, stores, etc.
     id is made opaque again at the end of the module.
  *)
  Transparent id.

  Section tests.
      Definition x1 : id := 1.
      Definition x2 : id := 2.
      Definition x3 : id := 3.
      Definition x4 : id := 4.
      Definition x5 : id := 5.

      Variables e1 e2 e3 e4 : expr.

      Definition c1 : command := cAssert e1.
      Definition c2 : command := cAssign x1 e2.
      Definition c3 : command := cSequence (cAssign x2 e3) (cAssert e3).
      Definition c4 : command := cSequence (cAssign x1 e1) (cAssign x2 e2).
      Definition c5 : command := cSequence (cAssign x1 e3) (cAssign x3 e4).
      Definition c6 : command := cChoice c4 c5.
      Definition c7 : command := cSequence (cAssign x1 e1) (cAssign x1 e2).

      Definition targets_c1 := Empty.
      Definition targets_c2 := singleton x1.
      Definition targets_c3 := singleton x2.
      Definition targets_c4 := union (singleton x1) (singleton x2).
      Definition targets_c5 := union (singleton x1) (singleton x3).
      Definition targets_c6 := union (union (singleton x1) (singleton x2))
                                    (union (singleton x1) (singleton x3)).

      Theorem test_targets_c1 : Equal (targets c1) empty.
      Proof. (* 1 lines *) Qed.

      Theorem test_targets_c2 : Equal (targets c2) targets_c2.
      Proof. (* 1 lines *) Qed.

      Theorem test_targets_c3 : Equal (targets c3) targets_c3.
      Proof. (* 6 lines *) Qed.
```

```
Theorem test_targets_c4 : Equal (targets c4) targets_c4.
Proof. (* 1 lines *) Qed.

Theorem test_targets_c5 : Equal (targets c5) targets_c5.
Proof. (* 1 lines *) Qed.

Theorem test_targets_c6 : Equal (targets c6) targets_c6.
Proof. (* 1 lines *) Qed.

Theorem test_fset_foldr_1 :
  forall n, fset_foldr nat (fun x y => 1) empty n = n.
Proof. (* 1 lines *) Qed.

Theorem test_fset_foldr_2 :
  forall n m, fset_foldr nat (fun x y => m) (singleton x1) n = m.
Proof. (* 1 lines *) Qed.

Theorem test_fset_foldr_3 :
  forall n, fset_foldr nat (fun x y => x + y) (singleton x1) n = S n.
Proof. (* 1 lines *) Qed.

Theorem test_fset_foldr_4 :
  forall n,
    fset_foldr nat (fun x y => x + y) targets_c6 n = x1 + x2 + x3 + n.
Proof. (* 1 lines *) Qed.

Definition v1 : vmap := fun _ => 0.

Definition v2 : vmap := fun _ => 5.

Definition v3 : vmap := fun x =>
  match x with
    | 1 => 5
    | 2 => 8
    | 3 => 1
    | 4 => 6
    | 5 => 3
    | _ => 0
  end.

Theorem test_insert_copy_vcmd_1 :
  forall c, insert_copy_vcmd v1 v1 x1 c = c.
Proof. (* 2 lines *) Qed.

Theorem test_insert_copy_vcmd_2 :
  forall c, insert_copy_vcmd v1 v2 x1 c = vcSequence (copy_vcmd x1 0 5) c.
Proof. (* 1 lines *) Qed.

Theorem test_insert_copy_vcmd_3 :
  forall c, insert_copy_vcmd v1 v3 x2 c = vcSequence (copy_vcmd x2 0 8) c.
Proof. (* 1 lines *) Qed.

Theorem test_insert_copy_vcmd_4 :
  forall c, insert_copy_vcmd v2 v3 x4 c = vcSequence (copy_vcmd x4 5 6) c.
Proof. (* 1 lines *) Qed.

Theorem test_sync_vcommand_1 :
  forall v v', sync_vcommand empty v v' = vcSkip.
Proof. (* 1 lines *) Qed.
```

```
      Theorem test_sync_vcommand_2 :
        sync_vcommand (singleton x1) v1 v2 =
          vcSequence (copy_vcmd x1 0 5) vcSkip.
      Proof. (* 1 lines *) Qed.

      Theorem test_sync_vcommand_3 :
        sync_vcommand targets_c6 v1 v3 =
          vcSequence (copy_vcmd x1 0 5)
                     (vcSequence (copy_vcmd x2 0 8)
                                 (vcSequence (copy_vcmd x3 0 1)
                                             vcSkip)).
      Proof. (* 1 lines *) Qed.

      Theorem test_sa_transformation_c1_c :
        let (c, v) := transform_sa c1 v1 in c = vcAssert (version_expr e1 v1).
      Proof. (* 1 lines *) Qed.

      Theorem test_sa_transformation_c1_v :
        let (c, v) := transform_sa c1 v1 in equivalent_functions v1 v.
      Proof. (* 1 lines *) Qed.

      Theorem test_sa_transformation_c2_c :
        let (c, v) := transform_sa c2 v1 in
          c = vcAssign (x1, 1) (version_expr e2 v1).
      Proof. (* 1 lines *) Qed.

      Theorem test_sa_transformation_c2_v :
        let (c, v) := transform_sa c2 v1 in
          equivalent_functions v (fun x => match x with 1 => 1 | _ => 0 end).
      Proof. (* 13 lines *) Qed.

      Theorem test_sa_transformation_c7_c :
        let (c, v) := transform_sa c7 v1 in
          c = vcSequence (vcAssign (x1, 1) (version_expr e1 v1))
                         (vcAssign (x1, 2) (version_expr e2 (inc v1 x1))).
      Proof. (* 3 lines *) Qed.

      Theorem test_sa_transformation_c7_v :
        let (c, v) := transform_sa c7 v1 in
          equivalent_functions v (fun x => match x with 1 => 2 | _ => 0 end).
      Proof. (* 10 lines *) Qed.

      Theorem test_sa_transformation_c6_c :
        let (c6', v) := transform_sa c6 v1 in
        let c4' := vcSequence (vcAssign (x1, 1) (version_expr e1 v1))
                              (vcAssign (x2, 1) (version_expr e2 (inc v1 x1))) in
        let d4  := vcSequence (copy_vcmd x3 0 1) vcSkip in
        let c5' := vcSequence (vcAssign (x1, 1) (version_expr e3 v1))
                              (vcAssign (x3, 1) (version_expr e4 (inc v1 x1))) in
        let d5  := vcSequence (copy_vcmd x2 0 1) vcSkip in
          c6' = vcChoice (vcSequence c4' d4) (vcSequence c5' d5).
      Proof. (* 2 lines *) Qed.
    End tests.

  Opaque id.
End Tests.
```

```
Lemma max_x_x :
  forall x, max x x = x.
Proof. (* 2 lines *) Qed.

(**
   [transform_sa] only updates the versions of those
   identifiers that are targets of [c]. This fact is
   important as it indicates the [vmap]s differ on
   a finite number of bindings. If this were not
   the case, it would be rather difficult to
   generate synchronization commands.
*)
Theorem transform_sa_vmap_delta :
  forall c c' v v',
    (c', v') = transform_sa c v -> vmap_delta v v' (targets c).
Proof. (* 36 lines *) Qed.

(**
   If mu and vmu are synchronized with respect to vmap v,
   then we can update x's binding in both mu and vmu so tha
   they are again synchronized, this time with
   respect to v', which is equivalent with v except
   that v' x = v x + 1 (i.e. x's version is incremented by one).

   Example:
«
   mu  = { x -> 5, y -> 3 }
   v   = { x -> 1, y -> 4 }
   mu' = { x_1 -> 5, y_4 -> 3 }

   updated mu   = { x -> 9, y -> 3 }
   updated v    = { x -> 2, y -> 4}
   updated mu'  = { x_1 -> 5, x_2 -> 9, y_4 -> 3 }
» *)
Lemma store_sync_vstore_assignment :
  forall mu vmu v e x,
    store_sync_vstore mu v vmu ->
    store_sync_vstore (update_store mu x (e mu))
                      (inc v x)
                      (update_vstore vmu (x, S (v x)) (version_expr e v vmu)).
Proof. (* 13 lines *) Qed.

Theorem vmap_delta_join_v_join :
  forall v v1 v2 D1 D2,
    vmap_delta v v1 D1 ->
    vmap_delta v v2 D2 ->
    vmap_delta v (join v1 v2) (IdSet.union D1 D2).
Proof. (* 5 lines *) Qed.

Theorem vmap_delta_join_v1_join :
  forall v v1 v2 D1 D2,
    vmap_delta v v1 D1 ->
    vmap_delta v v2 D2 ->
    vmap_delta v1 (join v1 v2) (IdSet.union D1 D2).
Proof. (* 4 lines *) Qed.
```

```
Theorem vmap_delta_join_v2_join :
  forall v v1 v2 D1 D2,
    vmap_delta v v1 D1 ->
    vmap_delta v v2 D2 ->
    vmap_delta v2 (join v1 v2) (IdSet.union D1 D2).
Proof. (* 4 lines *) Qed.

(**
   Given that [c'] is the single assignment form of [c],
   if [c] skips, so will [c'] (assuming the initial
   stores are synchronized).
   Also, both executions will end up in synchronized stores.
*)
Theorem sa_transformation_skip :
  forall c mu mu' vmu v,
    let (c', v') := transform_sa c v in
    multistep (ip c mu) (ip cSkip mu') ->
    store_sync_vstore mu v vmu ->
    exists vmu', vmultistep (vip c' vmu) (vip vcSkip vmu') /\
                 store_sync_vstore mu' v' vmu'.
Proof. (* 190 lines *) Qed.

(**
   If a command leads to failure, so will its single assignment form,
   if both are starting in synchronized stores.
*)
Theorem sa_transformation_fail :
  forall c mu mu' vmu v,
    let (c', v') := transform_sa c v in
      multistep (ip c mu) (failure mu') ->
      store_sync_vstore mu v vmu ->
      exists vmu', vmultistep (vip c' vmu) (vfailure vmu').
Proof. (* 97 lines *) Qed.

Inductive assigns (x : vid) : vcommand -> Prop :=
| assignsAssign    : forall e    ,                 assigns x (vcAssign x e)
| assignsSequenceL : forall c1 c2, assigns x c1 -> assigns x (vcSequence c1 c2)
| assignsSequenceR : forall c1 c2, assigns x c2 -> assigns x (vcSequence c1 c2)
| assignsChoiceL   : forall c1 c2, assigns x c1 -> assigns x (vcChoice c1 c2)
| assignsChoiceR   : forall c1 c2, assigns x c2 -> assigns x (vcChoice c1 c2).

Theorem assigns_dec :
  forall c x,
    decidable (assigns x c).
Proof. (* 24 lines *) Qed.

Inductive single_assignment_vid (x : vid) : vcommand -> Prop :=
| saidAssert    : forall e, single_assignment_vid x (vcAssert e)
| saidAssume    : forall e, single_assignment_vid x (vcAssume e)
| saidAssign    : forall y e, single_assignment_vid x (vcAssign y e)
| saidSequenceL : forall c1 c2,
                    ~ assigns x c1 -> single_assignment_vid x (vcSequence c1 c2)
| saidSequenceR : forall c1 c2,
                    ~ assigns x c2 -> single_assignment_vid x (vcSequence c1 c2)
| saidSkip      : single_assignment_vid x vcSkip
| saidChoice    : forall c1 c2,
                    single_assignment_vid x c1 ->
                    single_assignment_vid x c2 ->
                    single_assignment_vid x (vcChoice c1 c2).

Theorem single_assignment_vid_dec :
  forall c x, decidable (single_assignment_vid x c).
Proof. (* 14 lines *) Qed.

Inductive vmap_bound (v v' : vmap) : vcommand -> Prop :=
| vbAssert   : forall e, vmap_bound v v' (vcAssert e)
| vbAssume   : forall e, vmap_bound v v' (vcAssume e)
| vbAssign   : forall x n e,
               v x < n ->
               n <= v' x ->
               vmap_bound v v' (vcAssign (x, n) e)
| vbSequence : forall c1 c2,
               vmap_bound v v' c1 ->
               vmap_bound v v' c2 ->
               vmap_bound v v' (vcSequence c1 c2)
| vbSkip     : vmap_bound v v' vcSkip
| vbChoice   : forall c1 c2,
               vmap_bound v v' c1 ->
               vmap_bound v v' c2 ->
               vmap_bound v v' (vcChoice c1 c2).

Definition vmap_le (v v' : vmap) := forall x, v x <= v' x.

Lemma vmap_le_refl :
  forall v, vmap_le v v.
Proof. (* 1 lines *) Qed.

Hint Resolve vmap_le_refl.

Theorem sa_transformation_monotonic_vmap :
  forall c v,
    let (c', v') :=
      transform_sa c v in vmap_le v v'.
Proof. (* 35 lines *) Qed.

Lemma vmap_bound_le_upper :
  forall c v1 v2 v3,
    vmap_bound v1 v2 c ->
    vmap_le v2 v3 ->
    vmap_bound v1 v3 c.
Proof. (* 23 lines *) Qed.

Lemma vmap_bound_le_lower :
  forall c v1 v2 v3,
    vmap_le v1 v2 ->
    vmap_bound v2 v3 c ->
    vmap_bound v1 v3 c.
Proof. (* 19 lines *) Qed.

Lemma vmap_le_join_l :
  forall v1 v2, vmap_le v1 (join v1 v2).
Proof. (* 4 lines *) Qed.
```

```
Lemma vmap_le_join_r :
  forall v1 v2, vmap_le v2 (join v1 v2).
Proof. (* 5 lines *) Qed.

Lemma vmap_le_trans :
  forall v1 v2 v3, vmap_le v1 v2 -> vmap_le v2 v3 -> vmap_le v1 v3.
Proof. (* 2 lines *) Qed.

Theorem sync_vcommand_vmap_bound_l :
  forall D v1 v2,
    let joined := join v1 v2 in
      vmap_bound v1 joined (sync_vcommand D v1 joined).
Proof. (* 31 lines *) Qed.

Theorem sync_vcommand_vmap_bound_r :
  forall D v1 v2,
    let joined := join v1 v2 in
      vmap_bound v2 joined (sync_vcommand D v2 joined).
Proof. (* 31 lines *) Qed.

Theorem sa_transformation_vmap_bound :
  forall c v,
    let (c', v') := transform_sa c v in
      vmap_bound v v' c'.
Proof. (* 61 lines *) Qed.

Theorem assigns_vmap_bound :
  forall c x n v v',
    assigns (x, n) c ->
    vmap_bound v v' c ->
    v x < n /\ n <= v' x.
Proof. (* 20 lines *) Qed.

Theorem sa_transformation_is_single_assignment :
  forall c v x,
    let (c', _) := transform_sa c v in
      single_assignment_vid x c'.
Proof. (* 52 lines *) Qed.

(**
   ** Passification
*)

Definition assume_from_assign x e :=
  pcAssume (fun vmu => if decidable_eq_value (vmu x) (e vmu)
                     then T
                     else F).

Fixpoint passify (c : vcommand) : pcommand :=
  match c with
    | vcAssert e       => pcAssert e
    | vcAssume e       => pcAssume e
    | vcSkip           => pcSkip
    | vcSequence c1 c2 => pcSequence (passify c1) (passify c2)
    | vcChoice c1 c2   => pcChoice (passify c1) (passify c2)
    | vcAssign x e     => assume_from_assign x e
  end.
```

```
Definition stores_veq (vmu : vstore) (v : vmap) (vmu' : vstore) :=
  forall x n, n <= v x -> vmu (x, n) = vmu' (x, n).

Theorem vexpr_stores_veq :
  forall e v vmu vmu',
    stores_veq vmu v vmu' ->
    let ve := version_expr e v in
      ve vmu = ve vmu'.
Proof. (* 9 lines *) Qed.

Theorem stores_veq_refl :
  forall vmu v, stores_veq vmu v vmu.
Proof. (* 1 lines *) Qed.

Hint Resolve stores_veq_refl.

Theorem stores_veq_symm :
  forall vmu vmu' v, stores_veq vmu v vmu' -> stores_veq vmu' v vmu.
Proof. (* 2 lines *) Qed.

Hint Resolve stores_veq_symm.

Theorem stores_veq_trans :
  forall vmu vmu' vmu" v,
    stores_veq vmu v vmu' -> stores_veq vmu' v vmu" -> stores_veq vmu v vmu".
Proof. (* 4 lines *) Qed.

Theorem stores_veq_vmap_le :
  forall vmu vmu' v v',
    stores_veq vmu v vmu' -> vmap_le v' v -> stores_veq vmu v' vmu'.
Proof. (* 5 lines *) Qed.

Lemma stores_veq_sync_vcommand :
  forall D v1 v2 vmu vmu',
    vmap_le v1 v2 ->
    vmultistep (vip (sync_vcommand D v1 v2) vmu) (vip vcSkip vmu') ->
    stores_veq vmu v1 vmu'.
Proof. (* 48 lines *) Qed.

Theorem single_assignment_monotonic_store : forall c v vmu vmu',
  let (c', v') := transform_sa c v in
    vmultistep (vip c' vmu) (vip vcSkip vmu') -> stores_veq vmu v vmu'.
Proof. (* 91 lines *) Qed.

Theorem sync_vcommand_does_not_fail : forall D v v' vmu vmu',
  ~ vmultistep (vip (sync_vcommand D v v') vmu) (vfailure vmu').
Proof. (* 31 lines *) Qed.

Theorem single_assignment_monotonic_store_fail : forall c v vmu vmu',
  let (c', v') := transform_sa c v in
    vmultistep (vip c' vmu) (vfailure vmu') -> stores_veq vmu v vmu'.
Proof. (* 88 lines *) Qed.

Lemma sorted_list_x_lt_elts :
  forall (x : id) (xs : list id)
    (sorted_x_xs : sort (fun x y : id => x < y) (x :: xs)) (y : id),
    InA (fun x y : id => x = y) y xs -> x < y.
Proof. (* 28 lines *) Qed.
```

```
Lemma sorted_list_unique_elements :
  forall (x : id)
         (xs : list id)
         (sorted_x_xs : sort (fun x y : id => x < y) (x :: xs))
         (H : InA (fun x y : id => x = y) x xs), False.
Proof. (* 3 lines *) Qed.

Theorem vmultistep_pmultistep_sync_vcommand :
  forall D v v' vmu vmu' vmu",
    let c := sync_vcommand D v v' in
      vmap_le v v' ->
      vmultistep (vip c vmu) (vip vcSkip vmu') ->
      stores_veq vmu' v' vmu" ->
      pmultistep vmu" (pip (passify c)) (pip pcSkip).
Proof. (* 144 lines *) Qed.

Theorem vmultistep_pmultistep_skip : forall c v vmu vmu' vmu",
  let (c', v') := transform_sa c v in
    vmultistep (vip c' vmu) (vip vcSkip vmu') ->
    stores_veq vmu' v' vmu" ->
    pmultistep vmu" (pip (passify c')) (pip pcSkip).
Proof. (* 285 lines *) Qed.

(**
   Main theorem regarding passification:
   if the original program in its SA-form fails, so
   does its passification.
*)
Theorem vmultistep_pmultistep_fail :
  forall c v vmu vmu',
    let (c', v') := transform_sa c v in
      vmultistep (vip c' vmu) (vfailure vmu') ->
      pmultistep vmu' (pip (passify c')) pfailure.
Proof. (* 117 lines *) Qed.

(**
   ** Weakest Preconditions Soundness
*)
Fixpoint wp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
    | pcAssert e       => e vmu = T /\ Q
    | pcAssume e       => e vmu = T -> Q
    | pcChoice c1 c2   => wp vmu c1 Q /\ wp vmu c2 Q
    | pcSequence c1 c2 => wp vmu c1 (wp vmu c2 Q)
    | pcSkip           => Q
  end.

(**
   We show that if the weakest precondition holds,
   no single pstep will fail.
*)
Theorem pstep_wp_prevents_failure :
  forall (vmu : vstore) (c : pcommand) (Q : Prop),
    wp vmu c Q -> ~ pstep vmu (pip c) pfailure.
Proof. (* 17 lines *) Qed.
```

```
(**
   We prove that the weakest precondition is "preserved"
   along the pstep relation.
*)
Theorem pstep_wp_preservation :
  forall vmu c c' Q,
    wp vmu c Q -> pstep vmu (pip c) (pip c') -> wp vmu c' Q.
Proof. (* 40 lines *) Qed.

Theorem pmultistep_split_last :
  forall vmu s1 s3,
    pmultistep vmu s1 s3 ->
    s1 = s3 \/ exists s2, pmultistep vmu s1 s2 /\
                          pstep vmu s2 s3.
Proof. (* 13 lines *) Qed.

Theorem pmultistep_append :
  forall vmu s1 s2 s3,
    pmultistep vmu s1 s2 ->
    pstep vmu s2 s3 ->
    pmultistep vmu s1 s3.
Proof. (* 7 lines *) Qed.

Theorem pmultistep_forward_induction_scheme_aux
  (P : pstate -> pstate -> Prop)
  (vmu : vstore)
  (Hrefl : forall s, P s s)
  (Hstep : forall s1 s2 s3,
             pmultistep vmu s1 s2 -> pstep vmu s2 s3 -> P s1 s2 -> P s1 s3)
  (s1 s2 s3 : pstate)
  (H : P s1 s2)
  (H0 : pmultistep vmu s1 s2)
  (H1 : pmultistep vmu s2 s3) : P s1 s3.
Proof. (* 10 lines *) Qed.

Theorem pmultistep_forward_induction_scheme :
  forall
    (P : pstate -> pstate -> Prop)
    (vmu : vstore)
    (Hrefl : forall s, P s s)
    (Hstep : forall s1 s2 s3, pmultistep vmu s1 s2 ->
                              pstep vmu s2 s3 ->
                              P s1 s2 ->
                              P s1 s3),
    forall s1 s2, pmultistep vmu s1 s2 -> P s1 s2.
Proof. (* 3 lines *) Qed.

Theorem pmultistep_wp_preservation : forall vmu c1 c2 Q,
  wp vmu c1 Q -> pmultistep vmu (pip c1) (pip c2) -> wp vmu c2 Q.
Proof. (* 16 lines *) Qed.

Theorem pmultistep_wp_prevents_failure :
  forall vmu c Q,
    wp vmu c Q -> ~ pmultistep vmu (pip c) pfailure.
Proof. (* 17 lines *) Qed.
```

```
Fixpoint wlp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
    | pcAssert e       => e vmu = T -> Q
    | pcAssume e       => e vmu = T -> Q
    | pcChoice c1 c2   => wlp vmu c1 Q /\ wlp vmu c2 Q
    | pcSequence c1 c2 => wlp vmu c1 (wlp vmu c2 Q)
    | pcSkip           => Q
  end.

Theorem pstep_wlp_preservation :
  forall vmu c c' Q,
    wlp vmu c Q -> pstep vmu (pip c) (pip c') -> wlp vmu c' Q.
Proof. (* 40 lines *) Qed.

Theorem monotonic_wp :
  forall vmu c (Q R : Prop), (Q -> R) -> wp vmu c Q -> wp vmu c R.
Proof. (* 7 lines *) Qed.

Theorem monotonic_wlp :
  forall vmu c (Q R : Prop), (Q -> R) -> wlp vmu c Q -> wlp vmu c R.
Proof. (* 6 lines *) Qed.

Theorem wp_true :
  forall vmu c Q,
    wp vmu c Q -> wp vmu c True.
Proof. (* 1 lines *) Qed.

Theorem conjunctive_wp :
  forall vmu c Q R,
    wp vmu c Q /\ wp vmu c R <-> wp vmu c (Q /\ R).
Proof. (* 55 lines *) Qed.

Theorem conjunctive_wlp :
  forall vmu c Q R,
    wlp vmu c Q /\ wlp vmu c R <-> wlp vmu c (Q /\ R).
Proof. (* 58 lines *) Qed.

Theorem Q_impl_wlpQ :
  forall vmu c (Q : Prop), Q -> wlp vmu c Q.
Proof. (* 7 lines *) Qed.

Theorem wlp_true :
  forall vmu c, wlp vmu c True.
Proof. (* 1 lines *) Qed.

Theorem wlp_rewrite :
  forall vmu c Q, wlp vmu c Q <-> wlp vmu c False \/ Q.
Proof. (* 74 lines *) Qed.

Theorem wp_impl_wlp :
  forall vmu c Q, wp vmu c Q -> wlp vmu c Q.
Proof. (* 18 lines *) Qed.

Theorem wp_rewrite :
  forall vmu c Q,
    wp vmu c Q <-> wp vmu c True /\ wlp vmu c Q.
Proof. (* 37 lines *) Qed.
```

```
Fixpoint efficient_wlp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
    | pcAssert e       => e vmu = T -> Q
    | pcAssume e       => e vmu = T -> Q
    | pcSequence c1 c2 => efficient_wlp vmu c1 (efficient_wlp vmu c2 Q)
    | pcSkip           => Q
    | pcChoice c1 c2   => (efficient_wlp vmu c1 False /\
                          efficient_wlp vmu c2 False) \/ Q
  end.

Fixpoint efficient_wp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
    | pcAssert e       => e vmu = T /\ Q
    | pcAssume e       => e vmu = T -> Q
    | pcSequence c1 c2 => efficient_wp vmu c1 (efficient_wp vmu c2 Q)
    | pcSkip           => Q
    | pcChoice c1 c2   => efficient_wp vmu c1 True /\
                          efficient_wp vmu c2 True /\
                          efficient_wlp vmu (pcChoice c1 c2) Q
  end.

Theorem efficient_wlp_equivalence :
  forall vmu c Q, wlp vmu c Q <-> efficient_wlp vmu c Q.
Proof. (* 36 lines *) Qed.

Theorem efficient_wp_equivalence :
  forall vmu c Q, wp vmu c Q <-> efficient_wp vmu c Q.
Proof. (* 65 lines *) Qed.

(**
   Initial version map, where each identifier's version equals 0.
*)
Definition init_vmap (x : id) := 0.

(**
   Auxiliary definition to perform SA transformation and
   passification in one step.
*)
Definition passified (c : command) :=
  let (c', _) := transform_sa c init_vmap in
  passify c'.

(**
   Proves that for any store mu and
   version map v, there is a
   synchronized versioned store.
*)
Lemma versioned_store_exists :
  forall mu v, exists vmu, store_sync_vstore mu v vmu.
Proof. (* 6 lines *) Qed.

(**
   Proves the soundness of the efficient
   (conservative) weakest preconditions:
   If the weakest preconditions are true,
   execution will not encounter failure.
   #<font color="red">Needs manual checking
   (The theorem itself, not the proof)</font>#
*)
Theorem soundness_efficient_wp :
  forall c,
    (forall vmu, efficient_wp vmu (passified c) True) ->
    forall mu, ~ exists mu', multistep (ip c mu) (failure mu').
Proof. (* 23 lines *) Qed.

(** ** Weakest Preconditions Size *)

(**
   We can't measure the size of Props, so we define our own.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Inductive formula : Set :=
| fConjunction : formula -> formula -> formula
| fDisjunction : formula -> formula -> formula
| fImplication : formula -> formula -> formula
| fAtom        : formula.

(**
   We define a metric on formulae.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Fixpoint formula_metric (f : formula) : nat :=
  match f with
    | fConjunction x y => S (formula_metric x + formula_metric y)
    | fDisjunction x y => S (formula_metric x + formula_metric y)
    | fImplication x y => S (formula_metric x + formula_metric y)
    | fAtom            => 1
  end.

(**
   We define weakest liberal preconditions making use formula.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Fixpoint wlp' (c : pcommand) (Q : formula) {struct c} : formula :=
  match c with
    | pcAssert _       => fImplication fAtom Q
    | pcAssume _       => fImplication fAtom Q
    | pcSequence c1 c2 => wlp' c1 (wlp' c2 Q)
    | pcSkip           => fAtom
    | pcChoice c1 c2   => fDisjunction (fConjunction (wlp' c1 fAtom)
                                                     (wlp' c2 fAtom))
                                       Q
  end.
```

```
(**
   We define weakest preconditions making use formula.

   #<font color="red">Needs manual checking</font>#
   %{Needs manual checking}%
*)
Fixpoint wp' (c : pcommand) (Q : formula) {struct c} : formula :=
  match c with
    | pcAssert _       => fConjunction fAtom Q
    | pcAssume _       => fImplication fAtom Q
    | pcSequence c1 c2 => wp' c1 (wp' c2 Q)
    | pcSkip           => fAtom
    | pcChoice c1 c2   => fConjunction (fConjunction (wp' c1 fAtom)
                                                      (wp' c2 fAtom))
                                       (wlp' c Q)
  end.

(**
   Quick lemma showing that formulas are at least 1 big.
*)
Lemma formula_metric_ge_1 : forall Q, 1 <= formula_metric Q.
Proof. (* 1 lines *) Qed.

Hint Resolve formula_metric_ge_1.

(**
   Shows the linearity of the
   efficient weakest liberal preconditions
   with respect to the size of passified commands.
*)
Theorem linear_wlp' :
  exists a, forall c Q,
    formula_metric (wlp' c Q) <= a * pcommand_metric c + formula_metric Q.
Proof. (* 60 lines *) Qed.

(**
   Shows the quadracity of the
   efficient weakest preconditions with
   respect to passified commands
*)
Theorem quadratic_wp' :
  exists a, exists b, forall c Q,
    formula_metric (wp' c Q) <=
    a * pcommand_metric c * pcommand_metric c +
    b * pcommand_metric c + formula_metric Q.
Proof. (* 248 lines *) Qed.

(**
   A quick proof showing that passification results in a command
   which is the exact same size as its input.
*)
Theorem passify_maintains_size :
  forall c,
    vcommand_metric c = pcommand_metric (passify c).
Proof. (* 1 lines *) Qed.
```

```
(**
   A lemma showing that if x <= y, then x^2 <= y^2.
*)
Lemma monotonic_sqr :
  forall x y, x <= y -> x * x <= y * y.
Proof. (* 12 lines *) Qed.

(**
   Shows that the weakest preconditions are O(|c|^4 + |Q|).

   #<font color="red">Needs manual checking
   (The theorem itself, not the proof)</font>#
*)
Theorem polynomial_wps :
  exists N4, exists N3, exists N2, exists N1, forall c Q,
    let cp := passified c in
      let x := command_metric c in
        let wp := wp' cp Q in
          formula_metric wp <= N4 * x * x * x * x +
                               N3 * x * x * x +
                               N2 * x * x +
                               N1 * x +
                               formula_metric Q.
Proof. (* 67 lines *) Qed.
```

D.14 EnsembleExt

```
Require Export Ensembles.
Require Import Notations.
Require Import Utf8.
Require Import Misc.

Theorem Included_reflexivity : forall A (S : Ensemble A), S ⊆ S.
Proof. (* 4 lines *) Qed.

Theorem Included_transitivity :
  forall A (S S’ S” : Ensemble A), S ⊆ S’ → S’ ⊆ S” → S ⊆ S”.
Proof. (* 6 lines *) Qed.

Theorem Included_antisymmetric :
  forall A (S S’ : Ensemble A), S ⊆ S’ → S’ ⊆ S → Same_set _ S S’.
Proof. (* 3 lines *) Qed.

Theorem Same_set_reflexivity :
  forall A (B : Ensemble A), Same_set A B B.
Proof. (* 1 lines *) Qed.

Theorem Same_set_transitivity :
  forall (A : Type) (B C D : Ensemble A),
    Same_set _ B C → Same_set _ C D → Same_set _ B D.
Proof. (* 5 lines *) Qed.

Theorem Same_set_symmetry :
  forall (A : Type) (B C : Ensemble A),
    Same_set _ B C → Same_set _ C B.
Proof. (* 2 lines *) Qed.

Theorem empty_set_subseteq :
  forall (A : Type) (S : Ensemble A), ∅ ⊆ S.
Proof. (* 2 lines *) Qed.

Hint Immediate empty_set_subseteq.

Theorem Singleton_element :
  forall A (x y : A), x ∈ Singleton y ↔ x = y.
Proof. (* 3 lines *) Qed.

Theorem empty_empty_set :
  forall U (A : Ensemble U), (forall x, ˜ x ∈ A) ↔ Same_set _ A ∅.
Proof. (* 12 lines *) Qed.

Theorem union_A_A :
  forall U (A : Ensemble U), Same_set _ A (Union _ A A).
Proof. (* 4 lines *) Qed.

Theorem Union_commutative :
  forall U (A B : Ensemble U), Same_set _ (Union U A B) (Union U B A).
Proof. (* 12 lines *) Qed.

Inductive IndexedUnion {U I} (R : I → Ensemble U) : Ensemble U :=
  | IndexedUnion_intro :
      forall (i : I) (x : U) (H : x ∈ R i), x ∈ IndexedUnion R.
```

```
Theorem IndexedUnion_subsets :
  forall U I (R : I → Ensemble U) (i : I), R i ⊆ IndexedUnion R.
Proof. (* 2 lines *) Qed.

Theorem Union_with_subset :
  forall U (A A' : Ensemble U), A' ⊆ A -> Same_set U A (A ∪ A').
Proof. (* 11 lines *) Qed.

Definition IndexedIntersection {U I} (R : I → Ensemble U) : Ensemble U :=
  fun x => forall i : I, x ∈ R i.

Theorem IndexedIntersection_subset : forall U I (R : I → Ensemble U),
  forall i : I, IndexedIntersection R ⊆ R i.
Proof. (* 4 lines *) Qed.

Definition closed {U} (R : Ensemble (Ensemble U)) :=
  forall Σ Σ', Σ' ∈ R → Σ' ⊆ Σ → Σ ∈ R.

Theorem closed_IndexedUnion :
  forall {A I} (R : I → Ensemble (Ensemble A)),
    (forall i, closed (R i)) → closed (IndexedUnion R).
Proof. (* 4 lines *) Qed.

Theorem closed_IndexedIntersection :
  forall {A I} (R : I → Ensemble (Ensemble A)),
    (forall i, closed (R i)) → closed (IndexedIntersection R).
Proof. (* 2 lines *) Qed.

Theorem closed_empty_set : forall A, closed (@Empty_set (Ensemble A)).
Proof. (* 2 lines *) Qed.

Theorem IndexedUnion_in_closed :
  forall Σ I (f : I → Ensemble Σ) (R : Ensemble (Ensemble Σ)),
    inhabited I → closed R → (forall i : I, f i ∈ R) → IndexedUnion f ∈ R.
Proof. (* 5 lines *) Qed.

Definition Map {A B} (f : A → B) (S : Ensemble A) : Ensemble B :=
  fun b => exists a : From S, f (proj1_sig a) = b.
```

D.15 EqDec

```
Require Import Notations.

Module Type EQDEC.

  Parameters
    (t      : Set)
    (eq_dec : forall x y : t, { x = y } + { x ≠ y }).

End EQDEC.
```

D.16 Expression

```
Require Import Notations.
Require Import Arith.
Require Import Identifier.

Inductive t' : Set :=
| lit : nat   → t'
| var : Id.t  → t'
| add : t'    → t' → t'
| min : t'    → t' → t'
| mul : t'    → t' → t'.

Definition t := t'.

Fixpoint evaluate (expr : t) (s : Id.t → nat) : nat :=
  match expr with
    | lit n       => n
    | var x       => s x
    | add e e'    => evaluate e s + evaluate e' s
    | min e e'    => evaluate e s - evaluate e' s
    | mul e e'    => evaluate e s * evaluate e' s
  end.

Definition eq_dec : forall e e' : t, {e = e'} + {e ≠ e'} .
  induction e; destruct e'; try (right; discriminate).

  (* lit *)
  destruct (eq_nat_dec n n0); [ subst; left; trivial | right ].
  intro; elim n1.
  injection H; trivial.

  (* var *)
  destruct (Id.eq_dec t0 t1); [ subst; left; trivial | right ].
  intro; elim n.
  injection H; trivial.

  (* add *)
  destruct (IHe1 e'1); destruct (IHe2 e'2);
    try (subst; left; trivial; fail); right; subst.
  intro.
  injection H; intros; subst.
  elim n; trivial.
  intro H; injection H; intros; subst.
  elim n; trivial.
  intro H; injection H; intros; subst.
  elim n; trivial.

  (* min *)
  destruct (IHe1 e'1); destruct (IHe2 e'2);
    try (subst; left; trivial; fail); right; subst.
  intro.
  injection H; intros; subst.
  elim n; trivial.
  intro H; injection H; intros; subst.
  elim n; trivial.
  intro H; injection H; intros; subst.
  elim n; trivial.

  (* mul *)
  destruct (IHe1 e'1); destruct (IHe2 e'2);
    try (subst; left; trivial; fail); right; subst.
  intro.
  injection H; intros; subst.
  elim n; trivial.
  intro H; injection H; intros; subst.
  elim n; trivial.
  intro H; injection H; intros; subst.
  elim n; trivial.
  Defined.

  Definition beq (e e' : t) : bool.
  destruct (eq_dec e e').
  exact true.
  exact false.
Defined.

Module Notations.

  Coercion lit : nat  >-> t'.
  Coercion var : Id.t >-> t'.

  Delimit Scope expr_scope with expr.
  Bind Scope expr_scope with t.

  Infix "+" := add : expr_scope.
  Infix "-" := min : expr_scope.

  Open Scope expr_scope.

End Notations.
```

D.17 FakeModel

```
Require Import Notations.
Require Import Relations.
Require Import Misc.
Require Import EnsembleExt.
Require RADefinitions.
Require RANotations.
Require RAAxioms.
Require Import Classical.

Section State.

  Variable (Σ : Type).

  Inductive result : Type :=
    | singler (σ : Σ)                        : result
    | singleb (σ : Σ)                        : result
    | mul     (I : Type) (R : I → result) : result
    | add     (I : Type) (R : I → result) : result.

  Definition single := singler.

  Fixpoint models (S : Ensemble Σ) (R : result) : Prop :=
    match R with
      | mul J R     => forall i, models S (R i)
      | add J R     => exists i, models S (R i)
      | singler σ   => σ ∈ S
      | singleb σ   => σ ∈ S
    end.

  Definition impl (R R' : result) : Prop :=
    forall S, models S R → models S R'.

  Definition eqv (R R' : result) : Prop :=
    impl R R' ∧ impl R' R.

  Definition top : result.
    refine (mul False _).
    intro H; elim H.
  Defined.

  Definition bottom : result.
    refine (add False _).
    intro H; elim H.
  Defined.

  Theorem single_axiom : forall (s : Σ) (S : Ensemble Σ),
    models S (single s) ↔ s ∈ S.
  Proof. (* 1 lines *) Qed.

  Theorem top_axiom : forall (R : False → result),
    eqv top (mul False R).
  Proof. (* 1 lines *) Qed.

  Theorem bottom_axiom : forall (R : False → result),
    eqv bottom (add False R).
  Proof. (* 1 lines *) Qed.

  Theorem add_axiom : forall I (R : I → result) (S : Ensemble Σ),
    models S (add _ R) ↔ ∃ i, models S (R i).
  Proof. (* 1 lines *) Qed.

  Theorem mul_axiom : forall I (R : I → result) (S : Ensemble Σ),
    models S (mul _ R) ↔ ∀ i, models S (R i).
  Proof. (* 1 lines *) Qed.

  Theorem implies_axiom : forall (R R' : result),
    impl R R' ↔ forall (S : Ensemble Σ), models S R → models S R'.
  Proof. (* 1 lines *) Qed.

  Theorem monotonic_models_axiom :
    forall (R : result) (S S' : Ensemble Σ),
      S' ⊆ S → models S' R → models S R.
  Proof. (* 6 lines *) Qed.

  Fixpoint lift (op : Σ → result) (R : result) : result :=
    match R with
      | singler σ    => op σ
      | singleb σ    => bottom
      | add _ R      => add _ (fun i => lift op (R i))
      | mul _ R      => mul _ (fun i => lift op (R i))
    end.

  Theorem lift_add_axiom : forall (f : Σ → result) I (R : I → result),
    eqv (lift f (add _ R)) (add _ (fun i => lift f (R i))).
  Proof. (* 1 lines *) Qed.

  Theorem lift_mul_axiom : forall (f : Σ → result) I (R : I → result),
    eqv (lift f (mul _ R)) (mul _ (fun i => lift f (R i))).
  Proof. (* 1 lines *) Qed.

  Theorem lift_single_axiom :
    forall (f : Σ → result) (s : Σ),
      eqv (lift f (single s)) (f s).
  Proof. (* 1 lines *) Qed.

End State.

Ltac prove_antecedent_in H :=
  match goal with
    | [ X : ?Antecedent → _ |- _ ] =>
      match H with
        | X =>
          let A := fresh in
          assert (A : Antecedent); [ idtac | specialize (H A); clear A ]
      end
  end.

Theorem monotonic_lift_axiom_fails :
  ¬ forall (f g : nat → result nat) (R R' : result nat),
    (forall σ, impl _ (f σ) (g σ)) →
    impl _ R R' → impl _ (lift _ f R) (lift _ g R').
Proof. (* 15 lines *) Qed.
```

D.18 Formula

```
Require Import List.
Require Import Arith.
Require Import Notations.
Require Import Identifier.
Require Symbol.
Require Term.
Require BooleanExpression.

Module Symb := Symbol.Default.

Inductive t' : Set :=
| fla_eq  : Term.t' → Term.t → t'
| fla_lt  : Term.t' → Term.t → t'
| fla_and : t'      → t'     → t'
| fla_not : t'      → t'.

Definition t := t'.

Definition eq_dec : forall f f' : t, { f = f' } + { f ≠ f' } .
  induction f; destruct f'; try (right; discriminate; fail).
  destruct (Term.eq_dec t0 t2); destruct (Term.eq_dec t1 t3); subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.
  destruct (Term.eq_dec t0 t2); destruct (Term.eq_dec t1 t3); subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.
  destruct (IHf1 f'1); destruct (IHf2 f'2); subst; try (left; reflexivity; fail);
    right; intro; elim n; injection H; trivial.
  destruct (IHf f');
    [ left; subst; reflexivity | right; intro; elim n; injection H; trivial ].
Defined.

Definition beq (f f' : t) : bool.
  destruct (eq_dec f f'); [ exact true | exact false ].
Defined.

Fixpoint symbols (f : t) : list Symb.t :=
  match f with
    | fla_eq t1 t2   => Term.symbols t1 ++ Term.symbols t2
    | fla_lt t1 t2   => Term.symbols t1 ++ Term.symbols t2
    | fla_and f f'   => symbols f ++ symbols f'
    | fla_not f      => symbols f
  end.

Definition fla_or (p q : t) : t :=
  fla_not (fla_and (fla_not p) (fla_not q)).

Definition implies (φ φ' : t) : t :=
  fla_or (fla_not φ) φ'.

Definition fla_true : t :=
  fla_eq (Term.term_lit 0)
         (Term.term_lit 0).

Definition fla_false : t :=
  fla_not fla_true.
```

```
Fixpoint of_boolean_expression
  (b : BooleanExpression.t) (s : Id.t → Term.t) : t :=
  match b with
    | BooleanExpression.eq  e e' =>
      fla_eq (Term.of_expression e s) (Term.of_expression e' s)
    | BooleanExpression.lt  e e' =>
      fla_lt (Term.of_expression e s) (Term.of_expression e' s)
    | BooleanExpression.le  e e' =>
      fla_not (fla_lt (Term.of_expression e' s) (Term.of_expression e s))
    | BooleanExpression.and b b' =>
      fla_and (of_boolean_expression b s) (of_boolean_expression b' s)
    | BooleanExpression.not b =>
      fla_not (of_boolean_expression b s)
    | BooleanExpression.true =>
      fla_true
  end.

Fixpoint evaluate (φ : t) (s : Symb.t → nat) : bool :=
  match φ with
    | fla_eq t t'  => if eq_nat_dec (Term.evaluate t s) (Term.evaluate t' s)
                      then true
                      else false
    | fla_lt t t'  => if lt_dec (Term.evaluate t s) (Term.evaluate t' s)
                      then true
                      else false
    | fla_and φ φ' => andb (evaluate φ s) (evaluate φ' s)
    | fla_not φ    => negb (evaluate φ s)
  end.

Fixpoint metric (φ : t) : nat :=
  match φ with
    | fla_eq  t t' => S (Term.metric t + Term.metric t')
    | fla_lt  t t' => S (Term.metric t + Term.metric t')
    | fla_and φ φ' => S (metric φ + metric φ')
    | fla_not φ    => S (metric φ)
  end.

(*
   New induction scheme, i.e. on formula size
*)
Section InductionScheme.

  Variable (P : t → Prop).

  Hypothesis H :
    forall φ : t,
      (forall φ' : t, metric φ' < metric φ → P φ') → P φ.

  Definition Pn (n : nat) :=
    forall φ : t, metric φ <= n → P φ.

  Lemma P0 : Pn 0.
  Proof. (* 6 lines *) Qed.

  Lemma induction_step : forall n, Pn n → Pn (S n).
  Proof. (* 22 lines *) Qed.
  Theorem all_Pn : forall n : nat, Pn n.
  Proof. (* 3 lines *) Qed.

  Theorem induction_on_size : forall φ : t, P φ.
  Proof. (* 5 lines *) Qed.

End InductionScheme.

Module Notations.

  Infix "==" := fla_eq (at level 70) : formula_scope.
  Infix "<"  := fla_lt : formula_scope.
  Notation "¬ e" := (fla_not e) : formula_scope.
  Notation "f && f'" := (fla_and f f') : formula_scope.

  Delimit Scope formula_scope with fla.
  Bind Scope formula_scope with t.

End Notations.
```

D.19 Fresh

```
Require Import Notations.
Require Import List.

Module Type FRESH.

  Parameter t : Type.

  Parameter fresh : forall (xs : list t), { x : t | ¬ In x xs } .

End FRESH.
```

D.20 Heap

```
Require Import Notations.
Require Import ListExt.
Require Import Bool.
Require Import EqDec.
Require Import SetHolder.
Require Import Basics.

Module Type HEAP (Chunk : SET).

  Parameters
    (t        : Set)
    (empty    : t)
    (is_empty : t → bool)
    (produce  : Chunk.t → t → t)
    (consume  : Chunk.t → t → option t)
    (contains : Chunk.t → t → bool)
    (count    : Chunk.t → t → nat)
    (enum     : t → list Chunk.t)
    (union    : t → t → t).

  Axiom is_empty_empty : is_empty empty = true.

  Axiom contains_empty : forall c, contains c empty = false.

  Axiom enum_empty : enum empty = nil.

  Axiom empty_counts :
    forall h, is_empty h = true ↔ forall c, count c h = 0.

End HEAP.

Module ListHeap (Chunk : EQDEC) <: HEAP Chunk.

  Definition t : Set := list Chunk.t.

  Definition empty : t :=
    nil.

  Definition is_empty (heap : t) : bool :=
    match heap with
      | nil => true
      | _   => false
    end.

  Definition produce (chunk : Chunk.t) (heap : t) : t :=
    chunk :: heap.

  Definition contains (chunk : Chunk.t) (heap : t) : bool :=
    if in_dec Chunk.eq_dec chunk heap then true else false.

  Definition consume (chunk : Chunk.t) (heap : t) : option t :=
    if contains chunk heap
      then
        Some (remove Chunk.eq_dec chunk heap)
      else
        None.
  Definition enum (heap : t) : list Chunk.t :=
    heap.

  Definition union (h h' : t) : t :=
    app h h'.

  Fixpoint count (chunk : Chunk.t) (heap : t) : nat :=
    match heap with
      | nil   => 0
      | c::cs => if Chunk.eq_dec chunk c
                 then S (count chunk cs)
                 else count chunk cs
    end.

  Theorem is_empty_empty : is_empty empty = true.
  Proof. (* 1 lines *) Qed.

  Theorem contains_empty : forall c, contains c empty = false.
  Proof. (* 1 lines *) Qed.

  Theorem enum_empty : enum empty = nil.
  Proof. (* 1 lines *) Qed.

  Theorem empty_counts :
    forall h, is_empty h = true ↔ forall c, count c h = 0.
  Proof. (* 11 lines *) Qed.

End ListHeap.

Module Default (Chunk : EQDEC) := ListHeap Chunk.
```

D.21 Identifier

```
Require Import Notations.
Require Import Arith.

Module Type IDENTIFIER.
  Parameters
    (t      : Set)
    (eq_dec : forall x y : t, {x = y} + {x ≠ y}).
End IDENTIFIER.

Module Nat <: IDENTIFIER.

  Inductive t' : Set :=
  | Id : nat → t'.

  Definition t := t'.

  Definition eq_dec : forall x y : t, {x = y} + {x ≠ y} .
    intros x y; destruct x; destruct y.
    destruct (eq_nat_dec n n0); [ left; subst; trivial | right ].
    intro; elim n1.
    injection H; trivial.
  Defined.

  Definition beq (x y : t) : bool.
    destruct (eq_dec x y).
    exact true.
    exact false.
  Defined.

End Nat.

Module Id := Nat.
```

D.22 InductiveFormulae

```
Require Import Notations.
Require Import Relations.
Require Import Misc.
Require Import EnsembleExt.
Require RADefinitions.
Require RANotations.
Require RAAxioms.

Set Implicit Arguments.

Module InductiveFormulaeDefinitions
  <: RADefinitions.DEFINITIONS.

  Inductive formula (Σ : Type) : Type :=
  | f_and : forall I (R : I → formula Σ), formula Σ
  | f_or  : forall I (R : I → formula Σ), formula Σ
  | f_lit : Σ → formula Σ.

  Definition empty_R (Σ : Type) : False → formula Σ.
  intro H; destruct H.
  Defined.

  Definition f_true (Σ : Type) : formula Σ :=
    (@f_and Σ False (empty_R Σ)).

  Definition f_false (Σ : Type) : formula Σ :=
    (@f_or Σ False (empty_R Σ)).

  Definition ℛ (Σ : Type) : Type := formula Σ.

  Section StateSection.
    Variable Σ : Type.

    Definition single (x : Σ) : ℛ Σ := f_lit x.

    Fixpoint models (S : Ensemble Σ) (R : ℛ Σ) : Prop :=
      match R with
        | f_and J R' => forall j : J, models S (R' j)
        | f_or  J R' => exists j : J, models S (R' j)
        | f_lit s    => s ∈ S
      end.

    Definition add {I : Type} (R : I → ℛ Σ) : ℛ Σ := f_or R.

    Definition mul {I : Type} (R : I → ℛ Σ) : ℛ Σ := f_and R.

    Definition top : ℛ Σ := f_true Σ.

    Definition bottom : ℛ Σ := f_false Σ.

    Definition implies (R R' : ℛ Σ) : Prop :=
      forall S, models S R → models S R'.

    Definition equiv : relation (ℛ Σ) :=
      fun R R' => implies R R' ∧ implies R' R.
  End StateSection.
End InductiveFormulaeDefinitions.
```

```
Module InductiveFormulaeAxioms
        <: RAAxioms.AXIOMS InductiveFormulaeDefinitions.

  Import InductiveFormulaeDefinitions.

  Module RAN := RANotations.Make( InductiveFormulaeDefinitions ).
  Include RAN.

  Section StateSection.

    Variable Σ : Type.

    Theorem top_axiom :
      forall (R : False → 𝓡 Σ),
        ⊤ ⟺ ⊗ R.
    Proof. (* 8 lines *) Qed.

    Theorem bottom_axiom :
      forall (R : False → 𝓡 Σ),
        ⊥ ⟺ ⊕ R.
    Proof. (* 9 lines *) Qed.

    Theorem single_axiom :
      forall (s : Σ) (S : Ensemble Σ),
        S ⊨ single s ↔ s ∈ S.
    Proof. (* 3 lines *) Qed.

    Theorem add_axiom :
      forall I (R : I → 𝓡 Σ) (S : Ensemble Σ),
        S ⊨ ⊕ R ↔ ∃ i, S ⊨ R i.
    Proof. (* 8 lines *) Qed.

    Theorem mul_axiom :
      forall I (R : I → 𝓡 Σ) (S : Ensemble Σ),
        S ⊨ ⊗ R ↔ ∀ i, S ⊨ R i.
    Proof. (* 5 lines *) Qed.

    Theorem implies_axiom :
      forall (R R' : 𝓡 Σ),
        R ⇛ R' ↔ forall (S : Ensemble Σ), S ⊨ R → S ⊨ R'.
    Proof. (* 5 lines *) Qed.

    Theorem monotonic_models_axiom :
      forall (R : 𝓡 Σ) S S',
        S' ⊆ S → S' ⊨ R → S ⊨ R.
    Proof. (* 8 lines *) Qed.

  End StateSection.

End InductiveFormulaeAxioms.
```

D.23 ListExt

```
Require Import Notations.
Require Import Basics.
Require Export List.

Open Scope program_scope.

Fixpoint concatenate {A : Set} (xs : list (list A)) : list A :=
  match xs with
    | x::xs => x ++ concatenate xs
    | nil   => nil
  end.

Definition concat_map {A B : Set} (f : A -> list B) : list A -> list B :=
  concatenate ∘ map f.

Fixpoint distribute {A : Set} (xs : list (list (list A))) : list (list A) :=
  let aux :=
    fun xs ys => concat_map (fun x => map (fun y => x ++ y) ys) xs
    in
    fold_left aux xs (nil :: nil).

Definition FromList {A : Type} (xs : list A) : Type :=
  { x : A | In x xs } .
```

D.24 ListSetExt

```
Require Import Notations.
Require Import ListExt.
Require Export ListSet.

Definition set_existsb {A : Type} (pred : A → bool) (xs : set A) :=
  @existsb A pred xs.

Definition set_forallb {A : Type} (pred : A → bool) (xs : set A) :=
  @forallb A pred xs.

Definition set_exists {A : Type} (pred : A → Prop) (xs : set A) :=
  @Exists A pred xs.

Definition set_forall {A : Type} (pred : A → Prop) (xs : set A) :=
  @Forall A pred xs.
```

D.25 Misc

```
Require Import Notations.
Require Export FunctionalExtensionality.
Require Import Ensembles.
Require Import Relations.

Definition singleton (A : Type) := inhabited A ∧ forall a a' : A, a = a'.

Definition function_restriction {A B}
                                (f : A → B)
                                (P : A → Prop)
                                (x : { x | P x }) : B :=
  f (proj1_sig x).

Notation "f ↓ P" := (function_restriction f P) (at level 100).

Definition function_with_empty_domain A B (H : ¬ inhabited A) : A → B.
  intro a.
  elim H.
  constructor.
  exact a.
Qed.

Theorem unique_function_with_empty_domain :
  forall A B (H : ¬ inhabited A) (f : A → B),
    f = function_with_empty_domain A B H.
Proof. (* 6 lines *) Qed.

Definition From {U} (A : Ensemble U) : Type := { x | x ∈ A } .

Theorem function_restriction_union_l :
  forall U R (A B : Ensemble U) (f : U → R) (a : From A),
    let a_value : U := proj1_sig a
    in
    let a_in_A := proj2_sig a
    in
      (f ↓ A ∪ B) (exist _ a_value (Union_introl U A B a_value a_in_A)) =
      (f ↓ A) a.
Proof. (* 1 lines *) Qed.

Theorem function_restriction_union_r :
  forall U R (A B : Ensemble U) (f : U → R) (a : From B),
    let a_value : U := proj1_sig a in
    let a_in_B := proj2_sig a in
      (f ↓ A ∪ B) (exist _ a_value (Union_intror U A B a_value a_in_B)) =
      (f ↓ B) a.
Proof. (* 1 lines *) Qed.

Section BinaryRelation.

  Set Implicit Arguments.

  Variables (A B : Type) (P : relation B).

  Definition pointwise (f f' : A → B) : Prop :=
    forall a : A, P (f a) (f' a).
  Theorem pointwise_reflexivity :
    forall (H : reflexive _ P), reflexive _ pointwise.
  Proof. (* 2 lines *) Qed.

  Theorem pointwise_symmetry :
    forall (H : symmetric _ P), symmetric _ pointwise.
  Proof. (* 2 lines *) Qed.

  Theorem pointwise_transitivity :
    forall (H : transitive _ P), transitive _ pointwise.
  Proof. (* 2 lines *) Qed.

  Unset Implicit Arguments.

End BinaryRelation.
```

D.26 Nat

```
Require Import Notations.
Require Import Arith.
Require Import List.
Require Import MinMax.
Require Import Omega.

Definition t := nat.

Definition eq_dec := eq_nat_dec.

Definition max_list (xs : list t) : t :=
  fold_left max xs 0.

Theorem x_leq_fold_max :
  forall x y xs, x ≤ y → x ≤ fold_left max xs y.
Proof. (* 7 lines *) Qed.

Theorem xs_leq_fold_max :
  forall x y xs, In x xs → x ≤ fold_left max xs y.
Proof. (* 9 lines *) Qed.

Theorem xs_leq_max_list :
  forall x xs, In x xs → x ≤ max_list xs.
Proof. (* 1 lines *) Qed.

Theorem fold_max_in_xs :
  forall x xs,
    { fold_left max xs x = x } + { In (fold_left max xs x) xs } .
Proof. (* 15 lines *) Qed.

Theorem max_xs_in_xs :
  forall x xs, In (max_list (x::xs)) (x::xs).
Proof. (* 6 lines *) Qed.

Definition fresh (xs : list t) : { x : t | ¬ In x xs } .
  exists (S (max_list xs)).
  intro.
  induction xs.
  inversion H.
  assert (H0 := max_xs_in_xs a xs).
  assert (H1 := xs_leq_max_list (S (max_list (a::xs))) (a::xs) H).
  omega.
Defined.
```

D.27 Notations

```
Require Import Ensembles.
Require Import Basics.
Require Export Utf8.

(* Ensemble notations *)
Implicit Arguments Empty_set [U].
Implicit Arguments Singleton [U].
Implicit Arguments In [ U ].

Notation "∅" := (@Empty_set _).

Notation "x ∈ X" := (In X x) (at level 40).

Notation "A ⊆ B" := (Included _ A B) (at level 40).

Notation "A ∪ B" := (Union _ A B) (at level 50).

Notation "A ∩ B" := (Intersection _ A B) (at level 50).

Notation "□" := tt.
```

D.28 Predicate

```
Require Import Notations.
Require Import String.

Definition t := string.

Definition eq_dec := string_dec.

Definition beq (x y : t) :=
  if eq_dec x y then true else false.

Definition ptr := ("ptr")%string.

Definition mb  := ("mb")%string.
```

D.29 RAAxioms

```
Require Import Notations.
Require Import Ensembles.
Require RADefinitions.
Require RANotations.

Set Implicit Arguments.

Module Type AXIOMS
  (Import Definitions : RADefinitions.DEFINITIONS).

  Module RAN := RANotations.Make ( Definitions ).
  Include RAN.

  Section StateSection.

    Variable Σ : Type.

    Axiom single_axiom :
      forall (s : Σ) (S : Ensemble Σ),
        S ⊨ single s ↔ s ∈ S.

    Axiom top_axiom :
      forall (R : False → 𝓡 Σ),
        ⊤ ⟺ ⊗ R.

    Axiom bottom_axiom :
      forall (R : False → 𝓡 Σ),
        ⊥ ⟺ ⊕ R.

    Axiom add_axiom :
      forall I (R : I → 𝓡 Σ) (S : Ensemble Σ),
        S ⊨ ⊕ R ↔ ∃ i, S ⊨ R i.

    Axiom mul_axiom :
      forall I (R : I → 𝓡 Σ) (S : Ensemble Σ),
        S ⊨ ⊗ R ↔ ∀ i, S ⊨ R i.

    Axiom implies_axiom :
      forall (R R' : 𝓡 Σ),
        R ⇛ R' ↔ forall (S : Ensemble Σ),
                     S ⊨ R → S ⊨ R'.

    Axiom monotonic_models_axiom :
      forall (R : 𝓡 Σ) (S S' : Ensemble Σ),
        S' ⊆ S → S' ⊨ R → S ⊨ R.

  End StateSection.

  Ltac ra_axiom :=
    match goal with
      | |- models _ (add _)     => apply add_axiom
      | |- models _ (mul _)     => apply mul_axiom
      | |- models _ (single _)  => apply single_axiom
      | |- implies _ _          => apply implies_axiom
    end.
  Ltac ra_axiom_in H :=
    match goal with
      | [ H’ : models _ (add _) |- _ ] =>
        match H’ with
          | H => rewrite add_axiom in H
        end
      | [ H’ : models _ (mul _) |- _ ] =>
        match H’ with
          | H => rewrite mul_axiom in H
        end
      | [ H’ : models _ (single _) |- _ ] =>
        match H’ with
          | H => rewrite single_axiom in H
        end
      | [ H’ : implies _ _ |- _ ] =>
        match H’ with
          | H => rewrite implies_axiom in H
        end
    end.

  Ltac ra_axioms :=
    match goal with
      | [ H : _ |- _ ] => ra_axiom_in H; ra_axioms
      | _ => (ra_axiom; ra_axioms) || idtac
    end.

End AXIOMS.
```

D.30 RADefaultOperators

```
Require Import Notations.
Require Import Misc.
Require Import EnsembleExt.
Require Import ClassicalChoice.
Require RADefinitions.
Require RANotations.
Require RAAxioms.
Require RATheorems.
Require RAOperators.

Set Implicit Arguments.

Module Make
  (Import Definitions : RADefinitions.DEFINITIONS)
  (Import Axioms      : RAAxioms.AXIOMS Definitions)
    <: RAOperators.AXIOMS Definitions Axioms.

  Module RAN := RANotations.Make(Definitions).
  Include RAN.

  Module RAT := RATheorems.Make(Definitions)(Axioms).
  Import RAT.

  Section StateSection.

    Variable Σ Σ' : Type.

    Definition lift (f : Σ → 𝓡 Σ') (R : 𝓡 Σ) : 𝓡 Σ' :=
      ⊕ (fun S : { S | S ⊨ R } =>
        ⊗ (fun σ : From (proj1_sig S) =>
          f (proj1_sig σ))).

    Theorem monotonic_lift_axiom :
      forall (f g : Σ → 𝓡 Σ') (R R' : 𝓡 Σ),
        (forall Σ, f Σ ⭆ g Σ) → R ⭆ R' → lift f R ⭆ lift g R'.
    Proof. (* 20 lines *) Qed.

    Theorem lift_add_axiom :
      forall (f : Σ → 𝓡 Σ') I (R : I → 𝓡 Σ),
        lift f (add R) ⟺ add (fun i, lift f (R i)).
    Proof. (* 38 lines *) Qed.

    Theorem lift_single_axiom :
      forall (f : Σ → 𝓡 Σ') (s : Σ),
        lift f (single s) ⟺ f s.
    Proof. (* 25 lines *) Qed.

    Theorem lift_mul_axiom :
      forall (f : Σ → 𝓡 Σ') I (R : I → 𝓡 Σ),
        lift f (mul R) ⟺ mul (fun i, lift f (R i)).
    Proof. (* 53 lines *) Qed.

  End StateSection.

End Make.
```

D.31 RADefinitions

```
Require Import Notations.
Require Import Relations.
Require Import Ensembles.

Set Implicit Arguments.

Module Type DEFINITIONS.

  Parameter 𝓡 : Type → Type.

  Section StateSection.

    Variable Σ : Type.

    Parameters
      (single  : Σ → 𝓡 Σ)
      (models  : Ensemble Σ → 𝓡 Σ → Prop)
      (add     : forall {I : Type} (R : I → 𝓡 Σ), 𝓡 Σ)
      (mul     : forall {I : Type} (R : I → 𝓡 Σ), 𝓡 Σ)
      (top     : 𝓡 Σ)
      (bottom  : 𝓡 Σ)
      (implies : relation (𝓡 Σ)).

    Definition equiv : relation (𝓡 Σ) :=
      fun R R' => implies R R' ∧ implies R' R.

  End StateSection.

  Implicit Arguments top [ Σ ].
  Implicit Arguments bottom [ Σ ].

End DEFINITIONS.
```

D.32 RANotations

```
Require RADefinitions.

Module Make (D : RADefinitions.DEFINITIONS).

  Import D.

  Notation "⊕" := add.

  Notation "⊗" := mul.

  Notation "⊤" := (@top _).

  Notation "⊥" := (@bottom _).

  Notation "R ⇛ R'" := (implies R R') (at level 70).

  Notation "S ⊨ R" := (models S R) (at level 75).

  Notation "R ⟺ R'" := (equiv R R') (at level 70).

End Make.
```

D.33 RAOperatorTheorems

```
Require Import Notations.
Require Import Misc.
Require Import EnsembleExt.
Require Import Classical.
Require Import ClassicalChoice.
Require Import Setoid.
Require Import Basics.
Require Import FunctionalExtensionality.
Require RADefinitions.
Require RANotations.
Require RAAxioms.
Require RATheorems.
Require RAOperators.

Set Implicit Arguments.

Module Make
  (Import Definitions : RADefinitions.DEFINITIONS)
  (Import Axioms      : RAAxioms.AXIOMS Definitions)
  (Import Operators   : RAOperators.AXIOMS Definitions Axioms).

  Module RAN := RANotations.Make(Definitions).
  Import RAN.

  Module RAT := RATheorems.Make(Definitions)(Axioms).
  Import RAT.

  Module RAOP := RAOperators.Make(Definitions)(Axioms)(Operators).
  Import RAOP.

  Import RAOP.DoNotation.

  Theorem invariant_lift :
    forall Σ Σ' (f g : Σ → 𝓡 Σ') (R R' : 𝓡 Σ),
      (forall Σ, f Σ ⟺ g Σ) → R ⟺ R' → lift f R ⟺ lift g R'.
  Proof. (* 3 lines *) Qed.

  (* For some reason, we need to copy it here *)

  Add Parametric Relation Σ : (𝓡 Σ) (@implies Σ)
    reflexivity proved by (@implies_reflexivity Σ)
    transitivity proved by (@implies_transitivity Σ)
  as implies_relation'.

  Add Parametric Relation Σ : (𝓡 Σ) (@equiv Σ)
    reflexivity proved by (@equiv_reflexivity Σ)
    symmetry proved by (@equiv_symmetric Σ)
    transitivity proved by (@equiv_transitivity Σ)
  as equiv_relation'.

  Add Parametric Relation I Σ : (I → 𝓡 Σ) (@f_implies Σ I)
    reflexivity proved by (@f_implies_reflexivity Σ I)
    transitivity proved by (@f_implies_transitivity Σ I)
  as f_implies_relation'.
```

```
Add Parametric Relation I Σ : (I → ℛ Σ) (@f_equiv Σ I)
  reflexivity proved by (@f_equiv_reflexivity Σ I)
  symmetry proved by (@f_equiv_symmetric Σ I)
  transitivity proved by (@f_equiv_transitivity Σ I)
as f_equiv_relation'.

Add Parametric Morphism Σ (S : Ensemble Σ) : (models S) with
    signature (@equiv Σ) ==> impl
  as models_morphism_equiv'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ : (@models Σ) with
  signature (Included Σ) ++> (@implies Σ) ==> impl
as models_morphism'.
Proof. (* 5 lines *) Qed.

Add Parametric Morphism Σ I : (@add Σ I) with
  signature (@f_implies Σ I) ++> (@implies Σ)
as add_morphism_implies'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ I : (@mul Σ I) with
  signature (@f_implies Σ I) ++> (@implies Σ)
as mul_morphism_implies'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ I : (@add Σ I) with
  signature (@f_equiv Σ I) ++> (@equiv Σ)
as add_morphism_equiv'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ I : (@mul Σ I) with
  signature (@f_equiv Σ I) ++> (@equiv Σ)
as mul_morphism_equiv'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ Σ' : (@lift Σ Σ') with
  signature (@f_implies Σ' Σ) ==> (@implies Σ) ==> (@implies Σ')
as lift_morphism_implies'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ Σ' : (@lift Σ Σ') with
  signature (@f_equiv Σ' Σ) ==> (@equiv Σ) ==> (@equiv Σ')
as lift_morphism_equiv'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ : (@demonic Σ) with
  signature (@equiv Σ) ==> iff
as demonic_morphism_equiv'.
Proof. (* 5 lines *) Qed.

Add Parametric Morphism Σ : (@angelic Σ) with
  signature (@equiv Σ) ==> iff
as angelic_morphism_equiv'.
Proof. (* 5 lines *) Qed.
```

```
Theorem invariant_bind : forall {A B C : Type}
  (f f' : @primitive_operator A B) (g g' : @primitive_operator B C),
  f_equiv f f' → f_equiv g g' → f_equiv (f ⋙ g) (f' ⋙ g').
Proof. (* 2 lines *) Qed.

Add Parametric Morphism Σ Σ' Σ" : (@primitive_bind Σ Σ' Σ") with
  signature (@f_equiv Σ' Σ) ==> (@f_equiv Σ" Σ') ==> (@f_equiv Σ" Σ)
as bind_morphism_f_equiv.
Proof. (* 2 lines *) Qed.

Ltac lhs_equiv_rhs id id' :=
  match goal with
    | [ |- ?X ⟺ ?Y ] => set (id := X); set (id' := Y)
  end.

Ltac lift_axiom :=
  first
    [ rewrite lift_single_axiom
    | rewrite lift_add_axiom
    | rewrite lift_mul_axiom ].

Ltac lift_axiom_in H :=
  first
    [ rewrite lift_single_axiom in H
    | rewrite lift_add_axiom in H
    | rewrite lift_mul_axiom in H ].

Ltac lift_axioms :=
  match goal with
    | [ H : _ |- _ ] => lift_axiom_in H; lift_axioms
    | _              => (lift_axiom; lift_axioms) || idtac
  end.

Theorem bind_associative :
  forall {A B C D : Type}
    (f : @primitive_operator A B)
    (g : @primitive_operator B C)
    (h : @primitive_operator C D),
    f_equiv (f ⋙ (g ⋙ h)) ((f ⋙ g) ⋙ h).
Proof. (* 20 lines *) Qed.

Theorem lift_bottom :
  forall {A B : Type} (op : @primitive_operator A B),
    lift op ⊥ ⟺ ⊥.
Proof. (* 8 lines *) Qed.

Theorem lift_top :
  forall {A B : Type} (op : @primitive_operator A B),
    lift op ⊤ ⟺ ⊤.
Proof. (* 8 lines *) Qed.

Theorem lift_normalized :
  forall {A B : Type} (f : @primitive_operator A B) (R : 𝓡 A),
    lift f R ⟺ ⊕ (fun S : { S | S ⊨ R } =>
                 ⊗ (fun σ : From (proj1_sig S) =>
                   f (proj1_sig σ))).
Proof. (* 10 lines *) Qed.
```

```
Theorem lift_failure :
  forall A B R,
    ¬ R ⟺ ⊤ → lift (@failure A B) R ⟺ ⊥.
Proof. (* 37 lines *) Qed.

Theorem lifted_single :
  forall {A} (R : ℛ A),
    lift (@single A) R ⟺ R.
Proof. (* 15 lines *) Qed.

Theorem bind_top_left :
  forall {A B C : Type}
         (op    : @primitive_operator A B)
         (op'   : @primitive_operator B C)
         (a     : A),
    op a ⟺ ⊤ → (op ⋙ op') a ⟺ ⊤.
Proof. (* 3 lines *) Qed.

Theorem bind_bottom_left :
  forall {A B C : Type}
         (op    : @primitive_operator A B)
         (op'   : @primitive_operator B C)
         (a     : A),
    op a ⟺ ⊥ → (op ⋙ op') a ⟺ ⊥.
Proof. (* 3 lines *) Qed.

Theorem bind_failure_left :
  forall {A B C : Type} (op : @primitive_operator B C),
    f_equiv ((@failure A B) ⋙ op) (@failure A C).
Proof. (* 2 lines *) Qed.

Theorem bind_failure_right :
  forall {A B C : Type} (op : @primitive_operator A B)
         (a : A), ¬ (op a ⟺ ⊤) → (op ⋙ (@failure B C)) a ⟺ ⊥.
Proof. (* 30 lines *) Qed.

Theorem bind_block_right :
  forall {A B C : Type} (op : @primitive_operator A B)
         (a : A), ¬ (op a ⟺ ⊥) → (op ⋙ (@block B C)) a ⟺ ⊤.
Proof. (* 22 lines *) Qed.

Theorem bind_block_left :
  forall {A B C : Type} (op : @primitive_operator B C),
    f_equiv ((@block A B) ⋙ op) (@block A C).
Proof. (* 2 lines *) Qed.

Theorem bind_nop_left :
  forall {A B C : Type} (op : @operator A B C),
    f_equiv (nop »= fun _ => op) op.
Proof. (* 3 lines *) Qed.

Theorem bind_id_left :
  forall {A B : Type}
         (op : @primitive_operator A B),
    f_equiv (primitive_bind id op) (lift op).
Proof. (* 2 lines *) Qed.
```

```
Definition pure_op_K {A Σ : Type} (op : @operator Σ Σ A) (R : Σ → ℛ A) :=
  f_equiv (fun σ : Σ => lift (fun a : A => single (a, σ)) (R σ)) op.

Definition pure_op {A Σ : Type} (op : @operator Σ Σ A) :=
  exists R : Σ → ℛ A, pure_op_K op R.

Theorem normalisation_pure_op :
  forall (A Σ : Type) (op : @operator Σ Σ A),
    pure_op op ↔
    forall σ, op σ ⟺
      ⊕ (fun S : { S : Ensemble (A * Σ) | S ⊨ op σ } =>
        ⊗ (fun π : { π | π ∈ (proj1_sig S) } =>
            single (fst (proj1_sig π), σ))).
Proof. (* 66 lines *) Qed.

Theorem bind_pure : forall (A B Σ : Type)
  (op : @operator Σ Σ A) (f : A → @operator Σ Σ B),
  pure_op op → (forall a, pure_op (f a)) →
  pure_op (op »= f).
Proof. (* 24 lines *) Qed.

Definition independent_op_K
  {A Σ : Type}
  (op  : @operator Σ Σ A) (R   : ℛ A) :=
  f_equiv (fun σ : Σ => lift (fun a : A => single (a, σ)) R) op.

Definition independent_op {A Σ : Type} (op : @operator Σ Σ A) :=
  exists R : ℛ A, independent_op_K op R.

Theorem independent_implies_pure :
  forall (A Σ : Type) (op : @operator Σ Σ A),
    independent_op op → pure_op op.
Proof. (* 4 lines *) Qed.

Theorem normalisation_independent_op :
  forall (A Σ : Type) (op : @operator Σ Σ A),
    independent_op op ↔ exists R : ℛ A,
      forall σ, op σ ⟺
        ⊕ (fun S : { S : Ensemble A | S ⊨ R } =>
          ⊗ (fun a : { a | a ∈ proj1_sig S } =>
            single (proj1_sig a, σ))).
Proof. (* 55 lines *) Qed.

Theorem bind_independent :
  forall (A B Σ : Type)
         (op : @operator Σ Σ A)
         (f : A → @operator Σ Σ B),
    independent_op op →
    (forall a, independent_op (f a)) →
    independent_op (op »= f).
Proof. (* 23 lines *) Qed.

Definition angelic_op {A B : Type} (op : @primitive_operator A B) : Prop :=
  forall a : A, angelic (op a).

Definition demonic_op {A B : Type} (op : @primitive_operator A B) : Prop :=
  forall a : A, demonic (op a).
```

```
Definition deterministic_op {A B : Type}
                            (op : @primitive_operator A B) : Prop :=
  forall a : A, deterministic (op a).

Definition angelic_op_K
  {A B : Type}
  (op : @primitive_operator A B)
  (K : A → Ensemble B) : Prop :=
  forall σ : A, angelic_K (op σ) (K σ).

Definition demonic_op_K
  {A B : Type}
  (op : @primitive_operator A B)
  (K : A → Ensemble B) : Prop :=
  forall σ : A, demonic_K (op σ) (K σ).

Definition deterministic_op_K
  {A B : Type} (op : @primitive_operator A B)
  (K : A → B) : Prop :=
  forall σ : A, deterministic_K (op σ) (K σ).

Theorem bind_angelic_primitive :
  forall
    (Σ Σ' Σ" : Type)
    (op       : @primitive_operator Σ Σ')
    (op'      : @primitive_operator Σ' Σ"),
    angelic_op op → angelic_op op' → angelic_op (op ⋙ op').
Proof. (* 53 lines *) Qed.

Theorem bind_angelic :
  forall
    (Σ Σ' Σ" A B : Type)
    (op           : @operator Σ Σ' A)
    (f            : A → @operator Σ' Σ" B),
    angelic_op op → (forall a : A, angelic_op (f a)) → angelic_op (op »= f).
Proof. (* 6 lines *) Qed.

Theorem bind_demonic_primitive :
  forall
    (Σ Σ' Σ" : Type)
    (op       : @primitive_operator Σ Σ')
    (op'      : @primitive_operator Σ' Σ"),
    demonic_op op → demonic_op op' → demonic_op (op ⋙ op').
Proof. (* 57 lines *) Qed.

Theorem bind_demonic :
  forall
    (Σ Σ' Σ" A B : Type)
    (op           : @operator Σ Σ' A)
    (f            : A → @operator Σ' Σ" B),
    demonic_op op → (forall a : A, demonic_op (f a)) → demonic_op (op »= f).
Proof. (* 6 lines *) Qed.

Theorem det_iff_demang_op :
  forall (Σ Σ' : Type) (op : @primitive_operator Σ Σ'),
    deterministic_op op ↔ angelic_op op ∧ demonic_op op.
Proof. (* 6 lines *) Qed.
```

```
Theorem bind_deterministic_primitive :
  forall
    (Σ Σ' Σ” : Type)
    (op      : @primitive_operator Σ Σ')
    (op'     : @primitive_operator Σ' Σ”),
    deterministic_op op →
    deterministic_op op' →
    deterministic_op (op ⋙ op').
Proof. (* 5 lines *) Qed.

Theorem bind_deterministic :
  forall
    (Σ Σ' Σ” A B : Type)
    (op          : @operator Σ Σ' A)
    (f           : A → @operator Σ' Σ” B),
    deterministic_op op →
    (forall a : A, deterministic_op (f a)) →
    deterministic_op (op »= f).
Proof. (* 6 lines *) Qed.

Theorem deterministic_single :
  forall Σ : Type,
    deterministic_op (@single Σ).
Proof. (* 2 lines *) Qed.

Theorem demonic_op_compat_f_equiv :
  forall (Σ Σ' : Type) (op op' : @primitive_operator Σ Σ'),
    demonic_op op → f_equiv op op' → demonic_op op'.
Proof. (* 4 lines *) Qed.

Add Parametric Morphism Σ Σ' : demonic_op with
  signature (@f_equiv Σ Σ') ==> iff
  as demonic_op_morphism_f_equiv.
Proof. (* 7 lines *) Qed.

Theorem angelic_op_compat_f_equiv :
  forall (Σ Σ' : Type) (op op' : @primitive_operator Σ Σ'),
    angelic_op op → f_equiv op op' → angelic_op op'.
Proof. (* 4 lines *) Qed.

Add Parametric Morphism Σ Σ' : angelic_op with
  signature (@f_equiv Σ Σ') ==> iff
  as angelic_op_morphism_f_equiv.
Proof. (* 7 lines *) Qed.

Theorem deterministic_op_compat_f_equiv :
  forall (Σ Σ' : Type) (op op' : @primitive_operator Σ Σ'),
    deterministic_op op → f_equiv op op' → deterministic_op op'.
Proof. (* 3 lines *) Qed.

Add Parametric Morphism Σ Σ' : deterministic_op with
  signature (@f_equiv Σ Σ') ==> iff
  as deterministic_op_morphism_f_equiv.
Proof. (* 7 lines *) Qed.

Open Scope op_scope.
```

```
  Open Scope program_scope.

  Definition ra_map {A B} (f : A → B) (R : ℛ A) : ℛ B :=
    lift ((@single _) ∘ f) R.

  Theorem mapped_models :
    forall {A B} (R : ℛ A) (f : A → B) (S : Ensemble A),
      S ⊨ R → (Map f S) ⊨ (ra_map f R).
  Proof. (* 8 lines *) Qed.

  Close Scope program_scope.

  Close Scope op_scope.

  (* Monad laws *)

  Section MonadLaws.

    Variables (Σ Σ' Σ" Σ"' A B C : Type).

    Theorem monad_theorem_1 :
      forall (x : A) (f : A → @operator Σ Σ B),
        f_equiv (yield x »= f) (f x).
    Proof. (* 3 lines *) Qed.

    Theorem monad_theorem_2 :
      forall (op : @operator Σ Σ' A),
        f_equiv (op »= yield) op.
    Proof. (* 6 lines *) Qed.

    Theorem monad_theorem_3 :
      forall (m : @operator Σ Σ' A)
             (f : A → @operator Σ' Σ" B)
             (g : B → @operator Σ" Σ"' C),
      f_equiv ((m »= f) »= g) (m »= (fun x => (f x »=  g))).
    Proof. (* 12 lines *) Qed.

  End MonadLaws.

End Make.
```

D.34 RAOperators

```
Require Import Notations.
Require Import Ensembles.
Require Import Basics.
Require Import Classical.
Require RADefinitions.
Require RANotations.
Require RAAxioms.

Set Implicit Arguments.

Module Type AXIOMS
  (Import D : RADefinitions.DEFINITIONS)
  (Import A : RAAxioms.AXIOMS D).

  Module RAN := RANotations.Make(D).
  Include RAN.

  Section StateSection.
    Variables (Σ Σ' : Type).

    Parameter lift : (Σ → 𝓡 Σ') → 𝓡 Σ → 𝓡 Σ'.

    Axiom monotonic_lift_axiom :
      forall (f g : Σ → 𝓡 Σ') (R R' : 𝓡 Σ),
        (forall σ, f σ ⇛ g σ) → R ⇛ R' → lift f R ⇛ lift g R'.

    Axiom lift_add_axiom : forall (f : Σ → 𝓡 Σ') I (R : I → 𝓡 Σ),
        lift f (add R) ⟺ add (fun i => lift f (R i)).

    Axiom lift_mul_axiom : forall (f : Σ → 𝓡 Σ') I (R : I → 𝓡 Σ),
        lift f (mul R) ⟺ mul (fun i => lift f (R i)).

    Axiom lift_single_axiom : forall (f : Σ → 𝓡 Σ') (s : Σ),
        lift f (single s) ⟺ f s.
  End StateSection.
End AXIOMS.

Module Make
  (Import Definitions : RADefinitions.DEFINITIONS)
  (Import Axioms      : RAAxioms.AXIOMS Definitions)
  (Import Operators   : AXIOMS Definitions Axioms).

  Module RAN := RANotations.Make(Definitions). Include RAN.

  Definition primitive_operator (Σ Σ' : Type) := Σ → 𝓡 Σ'.

  Definition primitive_bind {Σ Σ' Σ" : Type}
    (op  : primitive_operator Σ Σ')
    (op' : primitive_operator Σ' Σ") : primitive_operator Σ Σ" :=
    compose (lift op') op.

  Definition operator (Σ Σ' A : Type) := primitive_operator Σ (A * Σ').

  Definition bind {Σ Σ' Σ" A B : Type}
    (op : operator Σ Σ' A) (f : A → operator Σ' Σ" B) :=
    primitive_bind op (prod_curry f).
```

```
Definition failure {Σ Σ' : Type} : primitive_operator Σ Σ' :=
  fun σ : Σ => @bottom Σ'.

Definition block {Σ Σ' : Type} : primitive_operator Σ Σ' :=
  fun σ : Σ => @top Σ'.

Definition pick_angelically {Σ : Type}
                            (A : Type) : operator Σ Σ A :=
  fun σ : Σ => add (fun a : A => single (a, σ)).

Definition pick_demonically {Σ : Type}
                            (A : Type) : operator Σ Σ A :=
  fun σ : Σ => mul (fun a : A => single (a, σ)).

Definition nop {Σ : Type} : operator Σ Σ unit :=
  fun σ : Σ => single (□, σ).

Definition assert {Σ : Type} (b : bool) : operator Σ Σ unit :=
  match b with
    | true  => nop
    | false => failure
  end.

Definition assume {Σ : Type} (b : bool) : operator Σ Σ unit :=
  match b with
    | true  => nop
    | false => block
  end.

Definition yield {Σ A : Type} (x : A) : operator Σ Σ A :=
  fun σ : Σ => single (x, σ).

Definition current_state {Σ : Type} : operator Σ Σ Σ :=
  fun σ : Σ => single (σ, σ).

Definition set_current_state {Σ Σ' : Type}
                             (σ' : Σ') : operator Σ Σ' unit :=
  fun σ : Σ => single (□, σ').

Module DoNotation.
  Delimit Scope op_scope with op.
  Bind Scope op_scope with operator.

  Notation "op ⋙ op'" :=
    (primitive_bind op op') (at level 40).

  Notation "op »= op'" :=
    (bind op op') (at level 40).

  Notation "x ← op ; rest" := (bind op (fun x => rest))
    (at level 100,
      op at level 99,
      rest at level 100,
      right associativity) : op_scope.

  Notation "op ; rest" :=
    (bind op (fun (_ : unit) => rest))
    (at level 100) : op_scope.
End DoNotation.
```

```
  Module Util.
    Open Scope program_scope.

    Import DoNotation.

    Section TypesSection.
      Variables (A B C : Type).

      Definition field := ((A → B) * (A → B → A))%type.

      Definition get (f : field) := fst f.

      Definition set (f : field) := snd f.

      Definition with_ (f : field) (g : B → C) := g ∘ get f.

      Definition update (f : field) (g : B → B) :=
        fun (s : A) => set f s (g (get f s)).
    End TypesSection.

    Section StateSection.
      Variable (Σ A B: Type).

      Open Scope op_scope.

      Definition current
        (f : field Σ A) : operator Σ Σ A :=
        σ ← current_state; yield (get f σ).

      Definition set_current
        (f : field Σ A) (x : A) : operator Σ Σ unit :=
        σ ← current_state;
        set_current_state (set f σ x).

      Definition with_current
        (f : field Σ A) (g : A → B) : operator Σ Σ B :=
        x ← current f; yield (g x).

      Definition update_current
        (f : field Σ A) (g : A → A) : operator Σ Σ unit :=
        x ← current f;
        set_current f (g x).

      Definition from_some (x : option A) : operator Σ Σ A :=
        match x with
          | Some x => yield x
          | None   => failure
        end.

      Close Scope op_scope.
    End StateSection.
  End Util.
End Make.
```

D.35 RATheorems

```
Require Import Notations.
Require Import Relations.
Require Import Ensembles.
Require Import Basics.
Require Import Setoid.
Require Import EnsembleExt.
Require Import Misc.
Require Import Classical.
Require Import ClassicalChoice.
Require RAAxioms.
Require RADefinitions.
Require RANotations.

Set Implicit Arguments.

Module Make
  (Import Definitions : RADefinitions.DEFINITIONS)
  (Import Axioms      : RAAxioms.AXIOMS Definitions).

  Import Definitions.
  Import Axioms.

  Module RAN := RANotations.Make ( Definitions ).
  Include RAN.

  Definition valid {Σ} (R : 𝓡 Σ) :=
    (fun _ => True) ⊨ R.

  Section StateSection.
    Variable Σ : Type.

    Definition f_implies {I} : relation (I → 𝓡 Σ) :=
      fun (R R' : I → 𝓡 Σ) =>
        ∀ i : I, implies (R i) (R' i).

    Definition f_equiv {I} : relation (I → 𝓡 Σ) :=
      fun (R R' : I → 𝓡 Σ) =>
        ∀ i : I, equiv (R i) (R' i).

    Definition invariant_function (f : 𝓡 Σ → 𝓡 Σ) :=
      ∀ R R' : 𝓡 Σ, R ⟺ R' → f R ⟺ f R'.

    Theorem implies_reflexivity :
      reflexive (𝓡 Σ) (@implies Σ).
    Proof. (* 3 lines *) Qed.

    Hint Immediate implies_reflexivity : radb.

    Theorem implies_transitivity :
      transitive (𝓡 Σ) (@implies Σ).
    Proof. (* 3 lines *) Qed.

    Add Parametric Relation : (𝓡 Σ) (@implies Σ)
      reflexivity proved by implies_reflexivity
      transitivity proved by implies_transitivity
    as implies_relation.
```

```
Theorem f_implies_reflexivity :
  forall I,
    reflexive (I → 𝓡 Σ) (@f_implies I).
Proof. (* 1 lines *) Qed.

Theorem f_implies_transitivity :
  forall I,
    transitive (I → 𝓡 Σ) (@f_implies I).
Proof. (* 3 lines *) Qed.

Add Parametric Relation I : (I → 𝓡 Σ) (@f_implies I)
  reflexivity proved by (@f_implies_reflexivity I)
  transitivity proved by (@f_implies_transitivity I)
as f_implies_relation.

Theorem equiv_reflexivity : reflexive (𝓡 Σ) (@equiv Σ).
Proof. (* 1 lines *) Qed.

Hint Immediate equiv_reflexivity : radb.

Hint Extern 1 =>
  unfold equiv in * |- * : radb.

Hint Extern 1 =>
  repeat match goal with
           | [ H : _ ∧ _ |- _ ] => destruct H
           | |- _ ∧ _ => split
         end : radb.

Theorem equiv_symmetric :
  symmetric (𝓡 Σ) (@equiv Σ).
Proof. (* 1 lines *) Qed.

Theorem equiv_transitivity :
  transitive (𝓡 Σ) (@equiv Σ).
Proof. (* 5 lines *) Qed.

Add Parametric Relation : (𝓡 Σ) (@equiv Σ)
  reflexivity proved by equiv_reflexivity
  symmetry proved by equiv_symmetric
  transitivity proved by equiv_transitivity
as equiv_relation.

Theorem f_equiv_reflexivity :
  forall I,
    reflexive (I → 𝓡 Σ) (@f_equiv I).
Proof. (* 1 lines *) Qed.

Hint Immediate f_equiv_reflexivity : radb.

Theorem f_equiv_symmetric :
  forall I,
    symmetric (I → 𝓡 Σ) (@f_equiv I).
Proof. (* 3 lines *) Qed.

Theorem f_equiv_transitivity :
  forall I,
    transitive (I → 𝓡 Σ) (@f_equiv I).
Proof. (* 3 lines *) Qed.
```

```
Add Parametric Relation I : (I → 𝓡 Σ) (@f_equiv I)
  reflexivity proved by (@f_equiv_reflexivity I)
  symmetry proved by (@f_equiv_symmetric I)
  transitivity proved by (@f_equiv_transitivity I)
as f_equiv_relation.

Theorem implies_compat_implies :
  forall (R1 R1' : 𝓡 Σ), R1' ⭆ R1 →
    forall (R2 R2' : 𝓡 Σ), R2 ⭆ R2' →
      R1 ⭆ R2 → R1' ⭆ R2'.
Proof. (* 3 lines *) Qed.

Theorem implies_compat_equiv :
  forall (R1 R1' : 𝓡 Σ), R1 ⟺ R1' →
    forall (R2 R2' : 𝓡 Σ), R2 ⟺ R2' →
      R1 ⭆ R2 → R1' ⭆ R2'.
Proof. (* 4 lines *) Qed.

Theorem models_compat_equiv :
  forall (R R' : 𝓡 Σ),
    R ⟺ R' ↔ (forall (S : Ensemble Σ), S ⊨ R ↔ S ⊨ R').
Proof. (* 13 lines *) Qed.

Theorem invariant_models :
  forall (R R' : 𝓡 Σ) S,
    R ⟺ R' → S ⊨ R → S ⊨ R'.
Proof. (* 2 lines *) Qed.

Add Parametric Morphism (S : Ensemble Σ) : (models S) with
  signature (@equiv Σ) ==> impl
as models_morphism_equiv.
Proof. (* 2 lines *) Qed.

Theorem normalization : forall (R : 𝓡 Σ),
  R ⟺ ⊕ (fun S : { S | S ⊨ R } =>
          ⊗ (fun σ : From (proj1_sig S) =>
              single (proj1_sig σ))).
Proof. (* 26 lines *) Qed.

Add Parametric Morphism : (@models Σ) with
  signature (Included Σ) ++> (@implies Σ) ==> impl
as models_morphism.
Proof. (* 5 lines *) Qed.

Theorem top_model : ∀ S : Ensemble Σ, S ⊨ ⊤.
Proof. (* 7 lines *) Qed.

Theorem bottom_model : ∀ S : Ensemble Σ, ¬(S ⊨ ⊥).
Proof. (* 7 lines *) Qed.

Theorem top_models_all : ∀ R : 𝓡 Σ, (∀ S, S ⊨ R) ↔ R ⟺ ⊤.
Proof. (* 4 lines *) Qed.

Theorem only_empty_set_models_top : ∀ R : 𝓡 Σ, ∅ ⊨ R ↔ R ⟺ ⊤.
Proof. (* 8 lines *) Qed.

Theorem bottom_implies_all : ∀ R : 𝓡 Σ, ⊥ ⭆ R.
Proof. (* 5 lines *) Qed.
```

```
Theorem only_bottom_without_models :
  ∀ R : 𝓡 Σ, (¬ ∃ S, S ⊨ R) ↔ R ⟺ ⊥.
Proof. (* 12 lines *) Qed.

Theorem all_implies_top :
  ∀ R : 𝓡 Σ, R ⇛ ⊤.
Proof. (* 3 lines *) Qed.

Theorem only_bottom_implies_bottom :
  ∀ R : 𝓡 Σ, R ⇛ ⊥ → R ⟺ ⊥.
Proof. (* 5 lines *) Qed.

Theorem top_implies_none :
  ∀ R : 𝓡 Σ, ⊤ ⇛ R → R ⟺ ⊤.
Proof. (* 5 lines *) Qed.

Theorem implies_single :
  forall (x : Σ) (R : 𝓡 Σ),
    R ⇛ single x ↔ (∀ S, S ⊨ R → x ∈ S).
Proof. (* 9 lines *) Qed.

Theorem add_singleton_index :
  forall I (R : I → 𝓡 Σ),
    singleton I → ∀ i : I, ⊕ R ⟺ R i.
Proof. (* 14 lines *) Qed.

Theorem mul_singleton_index :
  forall I (R : I → 𝓡 Σ),
    singleton I → ∀ i : I, ⊗ R ⟺ R i.
Proof. (* 11 lines *) Qed.

Theorem add_top :
  forall I (R : I → 𝓡 Σ),
    (∃ i : I, R i ⟺ ⊤) → ⊕ R ⟺ ⊤.
Proof. (* 11 lines *) Qed.

Theorem add_bottom : forall I (R : I → 𝓡 Σ),
    (∀ i : I, R i ⟺ ⊥) ↔ ⊕ R ⟺ ⊥.
Proof. (* 13 lines *) Qed.

Theorem mul_bottom : forall I (R : I → 𝓡 Σ),
    (∃ i : I, R i ⟺ ⊥) → ⊗ R ⟺ ⊥.
Proof. (* 13 lines *) Qed.

Theorem mul_top :
  forall I (R : I → 𝓡 Σ),
    (∀ i : I, R i ⟺ ⊤) ↔ ⊗ R ⟺ ⊤.
Proof. (* 13 lines *) Qed.

Theorem add_compat_implies :
  forall I (R R' : I → 𝓡 Σ),
    f_implies R R' → ⊕ R ⇛ ⊕ R'.
Proof. (* 8 lines *) Qed.

Add Parametric Morphism I : (@add Σ I) with
  signature f_implies ++> (@implies Σ)
as add_morphism_implies.
Proof. (* 2 lines *) Qed.
```

```
Theorem mul_compat_implies :
  forall I (R R' : I → 𝓡 Σ),
    f_implies R R' → ⊗ R ⇛ ⊗ R'.
Proof. (* 6 lines *) Qed.

Add Parametric Morphism I : (@mul Σ I) with
  signature f_implies ++> (@implies Σ)
as mul_morphism_implies.
Proof. (* 2 lines *) Qed.

Lemma f_equiv_to_f_impl :
  forall I (R R' : I → 𝓡 Σ),
    f_equiv R R' ↔ f_implies R R' ∧ f_implies R' R.
Proof. (* 7 lines *) Qed.

Theorem add_compat_equiv :
  forall I (R R' : I → 𝓡 Σ),
    f_equiv R R' → ⊕ R ⟺ ⊕ R'.
Proof. (* 9 lines *) Qed.

Add Parametric Morphism I : (@add Σ I) with
  signature f_equiv ++> (@equiv Σ)
as add_morphism_equiv.
Proof. (* 2 lines *) Qed.

Theorem mul_compat_equiv :
  forall I (R R' : I → 𝓡 Σ),
    f_equiv R R' → ⊗ R ⟺ ⊗ R'.
Proof. (* 8 lines *) Qed.

Add Parametric Morphism I : (@mul Σ I) with
  signature f_equiv ++> (@equiv Σ)
as mul_morphism_equiv.
Proof. (* 2 lines *) Qed.

Theorem R_implies_add :
  forall I (R : I → 𝓡 Σ) (i : I),
    R i ⇛ ⊕ R.
Proof. (* 5 lines *) Qed.

Theorem mul_implies_R :
  forall I (R : I → 𝓡 Σ) (i : I),
    ⊗ R ⇛ R i.
Proof. (* 5 lines *) Qed.

Theorem add_constant_R :
  forall I (R : I → 𝓡 Σ) (i : I),
    (∀ j, R i = R j) → ⊕ R ⟺ R i.
Proof. (* 13 lines *) Qed.

Theorem mul_constant_R :
  forall I (R : I → 𝓡 Σ) (i : I),
    (∀ j, R i = R j) → ⊗ R ⟺ R i.
Proof. (* 9 lines *) Qed.

Theorem forall_iff_mul : forall I (R : 𝓡 Σ) (R' : I → 𝓡 Σ),
  (forall i : I, R ⇛ R' i) ↔ R ⇛ ⊗ R'.
Proof. (* 17 lines *) Qed.
```

```
Theorem models_add_union : forall U (A B : Ensemble U) (R : U → ℛ Σ) S,
    S ⊨ ⊕ (R ↓ A ∪ B) ↔ S ⊨ ⊕ (R ↓ A) ∨ S ⊨ ⊕ (R ↓ B).
Proof. (* 39 lines *) Qed.

Theorem models_add_subset : forall U (A A' : Ensemble U) (R : U → ℛ Σ) S,
    A' ⊆ A → S ⊨ ⊕ (R ↓ A') → S ⊨ ⊕ (R ↓ A).
Proof. (* 7 lines *) Qed.

Theorem implies_add_unionl : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    ⊕ (R ↓ A) ⇛ ⊕ (R ↓ A ∪ B).
Proof. (* 5 lines *) Qed.

Theorem implies_add_unionr : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    ⊕ (R ↓ B) ⇛ ⊕ (R ↓ A ∪ B).
Proof. (* 4 lines *) Qed.

Theorem implies_add_subset : forall U (A A' : Ensemble U) (R : U → ℛ Σ),
    A' ⊆ A -> ⊕ (R ↓ A') ⇛ ⊕ (R ↓ A).
Proof. (* 5 lines *) Qed.

Theorem models_mul_union : forall U (A B : Ensemble U) (R : U → ℛ Σ) S,
    S ⊨ ⊗ (R ↓ A ∪ B) ↔ S ⊨ ⊗ (R ↓ A) ∧ S ⊨ ⊗ (R ↓ B).
Proof. (* 34 lines *) Qed.

Theorem models_mul_subset : forall U (A A' : Ensemble U) (R : U → ℛ Σ) S,
    A' ⊆ A → S ⊨ ⊗ (R ↓ A) → S ⊨ ⊗ (R ↓ A').
Proof. (* 7 lines *) Qed.

Theorem implies_mul_unionl :  forall U (A B : Ensemble U) (R : U → ℛ Σ),
    ⊗ (R ↓ A ∪ B) ⇛ ⊗ (R ↓ A).
Proof. (* 5 lines *) Qed.

Theorem implies_mul_unionr : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    ⊗ (R ↓ A ∪ B) ⇛ ⊗ (R ↓ B).
Proof. (* 4 lines *) Qed.

Theorem implies_mul_subset : forall U (A A' : Ensemble U) (R : U → ℛ Σ),
    A' ⊆ A -> ⊗ (R ↓ A) ⇛ ⊗ (R ↓ A').
Proof. (* 5 lines *) Qed.

Theorem add_bottoml : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    (forall (a : From A), (R ↓ A) a ⟺ ⊥) →
    ⊕ (R ↓ A ∪ B) ⟺ ⊕ (R ↓ B).
Proof. (* 27 lines *) Qed.

Theorem add_bottomr : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    (forall (b : From B), (R ↓ B) b ⟺ ⊥) →
    ⊕ (R ↓ A ∪ B) ⟺ ⊕ (R ↓ A).
Proof. (* 5 lines *) Qed.

Theorem mul_topl : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    (forall (a : From A), (R ↓ A) a ⟺ ⊤) →
    ⊗ (R ↓ A ∪ B) ⟺ ⊗ (R ↓ B).
Proof. (* 30 lines *) Qed.

Theorem mul_topr : forall U (A B : Ensemble U) (R : U → ℛ Σ),
    (forall (b : From B), (R ↓ B) b ⟺ ⊤) →
    ⊗ (R ↓ A ∪ B) ⟺ ⊗ (R ↓ A).
Proof. (* 5 lines *) Qed.
```

```
Theorem add_flatten :
  forall I J (R : I -> J -> 𝓡 Σ),
    ⊕ (fun i, ⊕ (R i)) ⟺ ⊕ (fun p => R (fst p) (snd p)).
Proof. (* 22 lines *) Qed.

Theorem add_flatten_dep :
  forall A I (J : I → Ensemble A) (R : A → 𝓡 Σ),
    ⊕ (fun i : I => ⊕ (R ↓ J i)) ⟺
      ⊕ (R ↓ IndexedUnion J).
Proof. (* 27 lines *) Qed.

Theorem mul_flatten :
  forall I J (R : I -> J -> 𝓡 Σ),
    ⊗ (fun i, ⊗ (R i)) ⟺ ⊗ (fun p => R (fst p) (snd p)).
Proof. (* 20 lines *) Qed.

Theorem mul_flatten_dep :
  forall A I (J : I → Ensemble A) (R : A → 𝓡 Σ),
    ⊗ (fun i : I => ⊗ (R ↓ J i)) ⟺
      ⊗ (R ↓ IndexedUnion J).
Proof. (* 29 lines *) Qed.

Theorem add_mul : forall I J (R : I → J → 𝓡 Σ),
  ⊕ (fun i, ⊗ (R i)) ⟺
    ⊗ (fun f, ⊕ (fun i, R i (f i))).
Proof. (* 60 lines *) Qed.

Theorem mul_add : forall I J (R : I → J → 𝓡 Σ),
  ⊗ (fun i, ⊕ (R i)) ⟺
    ⊕ (fun f : I → J => ⊗ (fun i, R i (f i))).
Proof. (* 42 lines *) Qed.

Theorem add_make_undep :
  forall (I J : Type) (R : J → 𝓡 Σ) (ι : I → Ensemble J) (i : I),
  ⊕ (fun j : From (ι i) => R (proj1_sig j)) ⟺
    ⊕ (fun j : J =>
      ⊕ (fun j' : From (Intersection _ (Singleton j) (ι i)) =>
        R (proj1_sig j'))).
Proof. (* 24 lines *) Qed.

Theorem swap_add :
  forall (I J : Type) (R : I → J → 𝓡 Σ),
    ⊕ (fun i : I => ⊕ (R i)) ⟺
      ⊕ (fun j : J => ⊕ (fun i : I => R i j)).
Proof. (* 10 lines *) Qed.

Theorem swap_mul :
  forall (I J : Type) (R : I → J → 𝓡 Σ),
    ⊗ (fun i : I => ⊗ (R i)) ⟺
      ⊗ (fun j : J => ⊗ (fun i : I => R i j)).
Proof. (* 5 lines *) Qed.

Theorem models_define_result :
  forall (R R' : 𝓡 Σ),
    (forall S : Ensemble Σ, S ⊨ R ↔ S ⊨ R') ↔ R ⟺ R'.
Proof. (* 3 lines *) Qed.
```

```
Definition angelic_K (R : ℛ Σ) (K : Ensemble Σ) : Prop :=
  R ⟺ ⊕ (fun s : From K => single (proj1_sig s)).

Definition demonic_K (R : ℛ Σ) (K : Ensemble Σ) : Prop :=
  R ⟺ ⊗ (fun s : From K => single (proj1_sig s)).

Definition deterministic_K (R : ℛ Σ) (K : Σ) : Prop :=
  R ⟺ single K.

Definition angelic (R : ℛ Σ) := exists S : Ensemble Σ, angelic_K R S.

Definition demonic (R : ℛ Σ) := exists S : Ensemble Σ, demonic_K R S.

Definition deterministic (R : ℛ Σ) := exists s : Σ, deterministic_K R s.

Theorem singleton_model_of_single :
  forall (s : Σ), Singleton s ⊨ single s.
Proof. (* 3 lines *) Qed.

Theorem deterministic_implies_angelic :
  forall (R : ℛ Σ), deterministic R → angelic R.
Proof. (* 21 lines *) Qed.

Theorem deterministic_implies_demonic :
  forall (R : ℛ Σ), deterministic R → demonic R.
Proof. (* 14 lines *) Qed.

Theorem angelic_singleton_models :
  forall (R : ℛ Σ), angelic R ↔
    forall S, S ⊨ R → exists s : Σ, s ∈ S ∧ Singleton s ⊨ R.
Proof. (* 30 lines *) Qed.

Theorem demonic_core_model :
  forall (R : ℛ Σ) (S : Ensemble Σ),
    demonic_K R S ↔ S ⊨ R ∧ ∀ S', S' ⊨ R → S ⊆ S'.
Proof. (* 33 lines *) Qed.

Theorem demonic_angelic_implies_deterministic :
  forall (R : ℛ Σ),
    angelic R → demonic R → deterministic R.
Proof. (* 21 lines *) Qed.

Add Parametric Morphism : (@implies Σ) with
  signature (@equiv Σ) ==> (@equiv Σ) ==> iff
as implies_morphism_equiv_equiv.
Proof. (* 5 lines *) Qed.

Theorem angelic_compat_equiv :
  forall R R' : ℛ Σ, angelic R → R ⟺ R' → angelic R'.
Proof. (* 6 lines *) Qed.

Theorem demonic_compat_equiv :
  forall R R' : ℛ Σ, demonic R → R ⟺ R' → demonic R'.
Proof. (* 4 lines *) Qed.

Add Parametric Morphism : demonic with
  signature (@equiv Σ) ==> iff
as demonic_morphism_equiv.
Proof. (* 5 lines *) Qed.
```

```
Add Parametric Morphism : angelic with
  signature (@equiv Σ) ==> iff
as angelic_morphism_equiv.
Proof. (* 5 lines *) Qed.

Theorem deterministic_compat_equiv :
  forall R R' : ℛ Σ,
    deterministic R → R ⟺ R' → deterministic R'.
Proof. (* 4 lines *) Qed.

Add Parametric Morphism : deterministic with
  signature (@equiv Σ) ==> iff
as deterministic_morphism_equiv.
Proof. (* 5 lines *) Qed.

Theorem angelic_K_compat_equiv :
  forall (R R' : ℛ Σ) (K : Ensemble Σ),
    R ⟺ R' → angelic_K R K → angelic_K R' K.
Proof. (* 3 lines *) Qed.

Theorem unique_angelic_K : forall (R : ℛ Σ) (K K' : Ensemble Σ),
    angelic_K R K → angelic_K R K' → Same_set _ K K'.
Proof. (* 35 lines *) Qed.

Theorem angelic_equiv_iff_same_charset :
  forall (R R' : ℛ Σ) (K K' : Ensemble Σ),
    angelic_K R K → angelic_K R' K' → (R ⟺ R' ↔ Same_set _ K K').
Proof. (* 10 lines *) Qed.

Add Parametric Morphism : angelic_K with
  signature (@equiv Σ) ==> (@Same_set Σ) ==> iff
as angelic_K_morphism.
Proof. (* 5 lines *) Qed.

Theorem demonic_K_compat_equiv :
  forall (R R' : ℛ Σ) (K : Ensemble Σ),
    R ⟺ R' → demonic_K R K → demonic_K R' K.
Proof. (* 3 lines *) Qed.

Theorem unique_demonic_K :
  forall (R : ℛ Σ) (K K' : Ensemble Σ),
    demonic_K R K → demonic_K R K' → Same_set _ K K'.
Proof. (* 32 lines *) Qed.

Theorem demonic_equiv_iff_same_charset :
  forall (R R' : ℛ Σ) (K K' : Ensemble Σ),
    demonic_K R K → demonic_K R' K' → (R ⟺ R' ↔ Same_set _ K K').
Proof. (* 10 lines *) Qed.

Add Parametric Morphism : demonic_K with
  signature (@equiv Σ) ==> (@Same_set Σ) ==> iff
as demonic_K_morphism.
Proof. (* 5 lines *) Qed.

Theorem deterministic_K_compat_equiv :
  forall (R R' : ℛ Σ) (K : Σ),
    R ⟺ R' → deterministic_K R K → deterministic_K R' K.
Proof. (* 3 lines *) Qed.
```

```
    Theorem unique_deterministic_K :
      forall (R : ℛ Σ) (K K' : Σ),
        deterministic_K R K → deterministic_K R K' → K = K'.
    Proof. (* 21 lines *) Qed.

    Theorem deterministic_equiv_iff_same_charset :
      forall (R R' : ℛ Σ) (K K' : Σ),
        deterministic_K R K → deterministic_K R' K' → (R ⟺ R' ↔ K = K').
    Proof. (* 10 lines *) Qed.

    Add Parametric Morphism : deterministic_K with
      signature (@equiv Σ) ==> eq ==> iff
    as deterministic_K_morphism.
    Proof. (* 5 lines *) Qed.
  End StateSection.

  Section PsiPhiSection.

    Variables
      (A B : Type)
      (ϕ : A → Ensemble B)
      (ψ : B → Ensemble A).

    Hypothesis inv : forall (a : A) (b : B), b ∈ ϕ a ↔ a ∈ ψ b.

    Theorem single_implies_mul_add_inv :
      forall a : A,
        single a ⇛
        ⊗ (fun b : From (ϕ a) =>
          ⊕ (fun a' : From (ψ (proj1_sig b)) =>
            single (proj1_sig a'))).
    Proof. (* 6 lines *) Qed.

    Theorem add_mul_inv_implies_single :
      forall b : B,
        ⊕ (fun a : From (ψ b) =>
          ⊗ (fun b' : From (ϕ (proj1_sig a)) =>
            single (proj1_sig b'))) ⇛ single b.
    Proof. (* 6 lines *) Qed.

  End PsiPhiSection.

End Make.
```

D.36 Routine

```
Require Import Notations.
Require Import String.

Module Type NAME.
  Parameters
    (t       : Set)
    (eq_dec  : forall x y : t, {x = y} + {x ≠ y}).
End NAME.

Module Name.

  Module String <: NAME.

    Definition t := string.

    Definition eq_dec := string_dec.

  End String.

End Name.

Module DefaultName := Name.String.
```

D.37 SIL

```
Require Import Notations.
Require Import Ensembles.
Require Import ListExt.
Require Import Bool.

Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Routine.
Require Predicate.

Module RName := Routine.DefaultName.

Inductive command : Set :=
  | cmd_skip
  | cmd_assign       (_ : Id.t)                 (_ : Expression.t)
  | cmd_read_heap    (_ : Id.t)                 (_ : Expression.t)
  | cmd_write_heap   (_ : Expression.t)         (_ : Expression.t)
  | cmd_sequence     (_ : command)              (_ : command)
  | cmd_malloc       (_ : Id.t)                 (_ : nat)
  | cmd_free         (_ : Expression.t)
  | cmd_if           (_ : BooleanExpression.t) (_ : command)      (_ : command)
  | cmd_routine_call (_ : RName.t)              (_ : Expression.t).

Record routine_definition : Set :=
  RoutineDefinition {
    argument      : Id.t;
    routine_body  : command
  } .

Record program : Set :=
  Program {
    routines      : RName.t → option routine_definition;
    main_routine  : RName.t
  } .

Inductive cmd_routine_reference : command → RName.t → Prop :=
  | crr_routine_call : forall r e,
      cmd_routine_reference (cmd_routine_call r e) r
  | crr_if_then :
    forall r b c1 c2, cmd_routine_reference c1 r →
      cmd_routine_reference (cmd_if b c1 c2) r
  | crr_if_else : forall r b c1 c2,
      cmd_routine_reference c2 r →
      cmd_routine_reference (cmd_if b c1 c2) r
  | crr_seq_left :
    forall r c1 c2,
      cmd_routine_reference c1 r →
      cmd_routine_reference (cmd_sequence c1 c2) r
  | crr_seq_right :
    forall r c1 c2,
      cmd_routine_reference c2 r →
      cmd_routine_reference (cmd_sequence c1 c2) r.
```

```
Definition defined_routines (p : program) : Ensemble RName.t :=
  fun rname => exists rdef, routines p rname = Some rdef.

Inductive program_routine_reference : program → RName.t → Prop :=
  | prr_intro : forall p r rname arg c,
    routines p r = Some (RoutineDefinition arg c) →
    cmd_routine_reference c rname →
    program_routine_reference p rname.

Definition routine_complete (p : program) : Prop :=
  forall r : RName.t, program_routine_reference p r → r ∈ defined_routines p.

Definition wellformed_program (p : program) : Prop :=
  routine_complete p.

Module Notations.

  Delimit Scope command_scope with command.

  Notation "'skip'" :=
    (cmd_skip) : command_scope.

  Notation "x := e" :=
    (cmd_assign x e) (at level 50) : command_scope.

  Notation "x := ⟦ y ⟧" :=
    (cmd_read_heap x y) (at level 50) : command_scope.

  Notation "⟦ x ⟧ := y" :=
    (cmd_write_heap x y) (at level 50, x ident, y ident) : command_scope.

  Notation "c ; d" :=
    (cmd_sequence c d) (at level 85) : command_scope.

  Notation "x := 'malloc' n" :=
    (cmd_malloc x n) (at level 50) : command_scope.

  Notation "'free' x" :=
    (cmd_free x) (at level 50) : command_scope.

  Notation "'If' b 'Then' c1 'Else' c2" :=
    (cmd_if b c1 c2) (at level 80) : command_scope.

  Notation "r [ x ]" :=
    (cmd_routine_call r x) (at level 50, x at level 0) : command_scope.

End Notations.
```

D.38 SILPP

```
Require Import Notations.
Require Import Ensembles.
Require Import ListExt.

Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Routine.
Require Predicate.
Require Assertion.
Require SIL.

Import Assertion.Notations.

Module RName := Routine.DefaultName.

Inductive command : Set :=
  | cmd_skip
  | cmd_assign       (_ : Id.t)         (_ : Expression.t)
  | cmd_read_heap    (_ : Id.t)         (_ : Expression.t)
  | cmd_write_heap   (_ : Expression.t) (_ : Expression.t)
  | cmd_sequence     (_ : command)      (_ : command)
  | cmd_malloc       (_ : Id.t)         (_ : nat)
  | cmd_free         (_ : Expression.t)
  | cmd_if           (_ : BooleanExpression.t) (_ : command) (_ : command)
  | cmd_routine_call (_ : RName.t)      (_ : Expression.t)
  | cmd_open         (_ : Predicate.t)  (_ : Expression.t)
  | cmd_close        (_ : Predicate.t)  (_ : Expression.t) (_ : Expression.t).

Fixpoint translate_command (c : command) : SIL.command :=
  match c with
    | cmd_skip              => SIL.cmd_skip
    | cmd_assign x e        => SIL.cmd_assign x e
    | cmd_read_heap x y     => SIL.cmd_read_heap x y
    | cmd_write_heap x y    => SIL.cmd_write_heap x y
    | cmd_malloc x n        => SIL.cmd_malloc x n
    | cmd_free x            => SIL.cmd_free x
    | cmd_routine_call r x  => SIL.cmd_routine_call r x
    | cmd_open  _ _         => SIL.cmd_skip
    | cmd_close _ _ _       => SIL.cmd_skip
    | cmd_if b c1 c2        =>
      SIL.cmd_if b (translate_command c1)
      (translate_command c2)
    | cmd_sequence c1 c2    =>
      SIL.cmd_sequence (translate_command c1) (translate_command c2)
  end.

Record predicate_definition : Set :=
  PredicateDefinition {
    argument_a       : Id.t;
    argument_b       : Id.t;
    predicate_body   : Assertion.t
  } .
```

```
Record routine_definition : Set :=
  RoutineDefinition {
    argument      : Id.t;
    precondition  : Assertion.t;
    postcondition : Assertion.t;
    routine_body  : command
  } .

Record program : Set :=
  Program {
    predicates     : Predicate.t  → option predicate_definition;
    routines       : RName.t      → option routine_definition;
    main_routine   : RName.t
  } .

Inductive cmd_routine_reference : command → RName.t → Prop :=
| crr_routine_call :
  forall r e,
    cmd_routine_reference (cmd_routine_call r e) r
| crr_if_then :
  forall r b c1 c2,
    cmd_routine_reference c1 r →
    cmd_routine_reference (cmd_if b c1 c2) r
| crr_if_else :
  forall r b c1 c2,
    cmd_routine_reference c2 r →
    cmd_routine_reference (cmd_if b c1 c2) r
| crr_seq_left :
  forall r c1 c2,
    cmd_routine_reference c1 r →
    cmd_routine_reference (cmd_sequence c1 c2) r
| crr_seq_right :
  forall r c1 c2,
    cmd_routine_reference c2 r →
    cmd_routine_reference (cmd_sequence c1 c2) r.

Inductive cmd_predicate_reference : command → Predicate.t → Prop :=
| cpr_open :
  forall p e,
    cmd_predicate_reference (cmd_open p e) p
| cpr_close :
  forall p e e',
    cmd_predicate_reference (cmd_close p e e') p
| cpr_if_then :
  forall p b c1 c2,
    cmd_predicate_reference c1 p →
    cmd_predicate_reference (cmd_if b c1 c2) p
| cpr_if_else :
  forall p b c1 c2,
    cmd_predicate_reference c2 p →
    cmd_predicate_reference (cmd_if b c1 c2) p
| cpr_seq_left :
  forall p c1 c2,
    cmd_predicate_reference c1 p →
    cmd_predicate_reference (cmd_sequence c1 c2) p
```

```
| cpr_seq_right :
  forall p c1 c2,
    cmd_predicate_reference c2 p →
    cmd_predicate_reference (cmd_sequence c1 c2) p.

Inductive routine_assertion_reference : routine_definition →
                                        Predicate.t →
                                        Prop :=
| rar_pre  :
  forall p arg pre post body,
    Assertion.assertion_predicate_reference pre p →
    routine_assertion_reference (RoutineDefinition arg pre post body) p
| rar_post :
  forall p arg pre post body,
    Assertion.assertion_predicate_reference post p →
    routine_assertion_reference (RoutineDefinition arg pre post body) p
| rar_body :
  forall p arg pre post body,
    cmd_predicate_reference body p →
    routine_assertion_reference (RoutineDefinition arg pre post body) p.

Definition defined_routines (p : program) : Ensemble RName.t :=
  fun rname => exists rdef, routines p rname = Some rdef.

Definition defined_predicates (p : program) : Ensemble Predicate.t :=
  fun pred => exists pdef, predicates p pred = Some pdef.

Inductive program_routine_reference : program → RName.t → Prop :=
  | prr_intro :
    forall p r rname arg pre post c,
      routines p r = Some (RoutineDefinition arg pre post c) →
      cmd_routine_reference c rname →
      program_routine_reference p rname.

Inductive program_assertion_reference : program → Predicate.t → Prop :=
  | par_routine :
    forall p rname rdef pred,
      routines p rname = Some rdef →
      routine_assertion_reference rdef pred →
      program_assertion_reference p pred
  | par_preddef :
    forall p pname x y body,
      predicates p pname = Some (PredicateDefinition x y body) →
      Assertion.assertion_predicate_reference body pname →
      program_assertion_reference p pname.

Definition routine_complete (p : program) : Prop :=
  forall r : RName.t,
    program_routine_reference p r → r ∈ defined_routines p.

Definition predicate_complete (p : program) : Prop :=
  forall pred : Predicate.t,
    program_assertion_reference p pred → pred ∈ defined_predicates p.

Definition wellformed_program (p : program) : Prop :=
  routine_complete p ∧ predicate_complete p.
```

```
Definition translate_program (p : program) : SIL.program :=
  let aux :=
    fun r => match routines p r with
               | Some (RoutineDefinition arg _ _ body) =>
                 Some (SIL.RoutineDefinition arg (translate_command body))
               | None =>
                 None
             end
  in
  SIL.Program aux (main_routine p).

Module Notations.

  Delimit Scope command_scope with command.

  Notation "'skip'" :=
    (cmd_skip) : command_scope.

  Notation "x := e" :=
    (cmd_assign x e) (at level 50) : command_scope.

  Notation "x := ⟦ e ⟧" :=
    (cmd_read_heap x e) (at level 50) : command_scope.

  Notation "⟦ e ⟧ := e'" :=
    (cmd_write_heap e e') (at level 50) : command_scope.

  Notation "c ; d" :=
    (cmd_sequence c d) (at level 85) : command_scope.

  Notation "x := 'malloc' n" :=
    (cmd_malloc x n) (at level 50) : command_scope.

  Notation "'free' x" :=
    (cmd_free x) (at level 50) : command_scope.

  Notation "'If' b 'Then' c1 'Else' c2" :=
    (cmd_if b c1 c2) (at level 80) : command_scope.

  Notation "r [ x ]" :=
    (cmd_routine_call r x) (at level 50, x at level 0) : command_scope.

  Notation "'open' x [ e ; ? ]" :=
    (cmd_open x e) (x at level 10, e at level 0) : command_scope.

  Notation "'close' x [ e1 ; e2 ]" :=
    (cmd_close x e1 e2) (x at level 20, e1 at level 0, e2 at level 0) :
command_scope.

  Open Scope command_scope.

End Notations.
```

D.39 SMT

```
Require Import List.
Require Import Arith.
Require Import Notations.
Require Formula.
Require Symbol.
Require Term.

Ltac prove_antecedent H :=
  match goal with
    | [ H' : ?X → _ |- _ ] =>
      match H with
        H' => let aux := fresh in
                assert (aux : X);
                [ idtac | specialize (H aux); clear aux ]
      end
  end.

Module Symb := Symbol.Default.

Inductive result : Set :=
| Unsat
| Sat
| NoClue
| Valid.

Theorem result_dec_eq : forall r r' : result, { r = r' } + { r ≠ r' } .
Proof. (* 2 lines *) Qed.

Definition result_beq (r r' : result) : bool :=
  if result_dec_eq r r' then true else false.

Definition models (I : Symb.t → nat) (φ : Formula.t) : Prop :=
  Formula.evaluate φ I = true.

Definition has_model (φ : Formula.t) : Prop :=
  exists I : Symb.t → nat, models I φ.

Definition valid (φ : Formula.t) : Prop := forall I : Symb.t → nat, models I φ.

Theorem valid_implies_has_model : forall φ, valid φ → has_model φ.
Proof. (* 4 lines *) Qed.

Theorem valid_implies_model : forall φ I, valid φ → models I φ.
Proof. (* 1 lines *) Qed.

Theorem conjunction_models :
  forall I φ φ',
    models I (Formula.fla_and φ φ') ↔ models I φ ∧ models I φ'.
Proof. (* 12 lines *) Qed.

Theorem negconj_models :
  forall I φ φ',
    models I (Formula.fla_not (Formula.fla_and φ φ')) ↔
    models I (Formula.fla_not φ) ∨ models I (Formula.fla_not φ').
Proof. (* 15 lines *) Qed.
```

```
Theorem negneg_models :
  forall I φ,
    models I φ ↔ models I (Formula.fla_not (Formula.fla_not φ)).
Proof. (* 8 lines *) Qed.

Module Type SOLVER.
  Parameter decide : Formula.t → result.

  Parameter get_value : Formula.t → Term.t → option nat.

  Axiom decide_soundness_sat : forall φ : Formula.t,
    decide φ = Sat → has_model φ.

  Axiom decide_soundness_unsat : forall φ : Formula.t,
    decide φ = Unsat → ¬ has_model φ.

  Axiom decide_soundness_valid : forall φ : Formula.t,
    decide φ = Valid → valid φ.
End SOLVER.

Module MakeClueless <: SOLVER.
  Definition decide (φ : Formula.t) : result :=
    NoClue.

  Definition get_value (φ : Formula.t) (t : Term.t) : option nat :=
    None.

  Theorem decide_soundness_sat :
    forall φ : Formula.t,
      decide φ = Sat → has_model φ.
  Proof. (* 1 lines *) Qed.

  Theorem decide_soundness_unsat : forall φ : Formula.t,
    decide φ = Unsat → ¬ has_model φ.
  Proof. (* 1 lines *) Qed.

  Theorem decide_soundness_valid : forall φ : Formula.t,
    decide φ = Valid → valid φ.
  Proof. (* 1 lines *) Qed.

End MakeClueless.

Module MakeClueful.
  Import Formula.

  Definition context := list Formula.t.

  Definition operator A := context → list (A * context).

  Definition bind {A B : Type}
                  (op : operator A)
                  (f : A → operator B) : operator B :=
    fun ctx =>
      let results := op ctx in
       flat_map (fun pair : A * context =>
                  let (r, ctx) := pair in f r ctx)
                results.

  Definition get_context : operator context :=
    fun s => (s, s) :: nil.
```

```
Definition set_context (ctx : context) : operator unit :=
  fun _ => (tt, ctx) :: nil.

Notation "op ⋙ f" :=
  (bind op f) (at level 40).

Notation "x ← op ; f" :=
  (bind op (fun x => f)) (at level 40).

Definition yield {A : Type} (r : A) : operator A :=
  fun ctx => (r, ctx) :: nil.

Definition in_context (ϕ : Formula.t) : operator bool :=
  ctx ← get_context;
  yield (if in_dec Formula.eq_dec ϕ ctx then true else false).

Definition fail {A : Type} : operator A :=
  fun _ => nil.

Definition add_to_context (ϕ : Formula.t) : operator unit :=
  ctx ← get_context;
  set_context (ϕ :: ctx).

Definition split {A : Type} (op op' : operator A) :=
  fun ctx => op ctx ++ op' ctx.

Notation "op ⊙ op'" := (split op op') (at level 80).

Fixpoint assume (ϕ : Formula.t) : operator unit :=
  match ϕ with
    | fla_eq t t'  => r ← in_context ϕ;
                      if r
                        then yield tt
                        else r' ← in_context (fla_not ϕ);
                             if r
                               then fail
                               else if Term.eq_dec t t'
                                    then yield tt
                                    else add_to_context ϕ
    | fla_lt t t'  => r ← in_context ϕ;
                      if r
                        then yield tt
                        else r' ← in_context (fla_not ϕ);
                             if r
                               then fail
                               else if Term.eq_dec t t'
                                    then fail
                                    else add_to_context ϕ
    | fla_and ϕ ϕ' => _ ← assume ϕ;
                      assume ϕ'
    | fla_not ϕ => assume_false ϕ
  end
    with assume_false (ϕ : Formula.t) : operator unit :=
      match ϕ with
        | fla_not ϕ => assume ϕ
        | fla_and ϕ ϕ' => assume_false ϕ ⊙ assume_false ϕ'
        | fla_eq t t' => r ← in_context ϕ;
```

```
                         if r
                           then fail
                           else r' ← in_context (fla_not φ);
                                if r
                                  then yield tt
                                  else if Term.eq_dec t t'
                                       then fail
                                       else add_to_context (fla_not φ)
        | fla_lt t t' => r ← in_context φ;
                         if r
                           then fail
                           else r' ← in_context (fla_not φ);
                                if r
                                  then yield tt
                                  else if Term.eq_dec t t'
                                       then yield tt
                                       else add_to_context (fla_not φ)
      end.

Definition decide (φ : Formula.t) : result :=
  match assume φ nil with
    | nil => Unsat
    | _   => NoClue
  end.

Fixpoint get_value_aux (φs : list Formula.t) (t : Term.t) : option nat :=
  match φs with
    | nil   => None
    | φ::φs => match φ with
                 | fla_eq t' (Term.term_lit n) => if Term.eq_dec t t'
                                                  then Some n
                                                  else get_value_aux φs t
                 | fla_eq (Term.term_lit n) t' => if Term.eq_dec t t'
                                                  then Some n
                                                  else get_value_aux φs t
                 | _ => get_value_aux φs t
               end
  end.

Definition get_value (φ : Formula.t) (t : Term.t) : option nat :=
  match t with
    | Term.term_lit n => Some n
    |  _              => match assume φ nil with
                           | ctx :: nil => get_value_aux (snd ctx) t
                           | _          => None
                         end
  end.

Theorem decide_soundness_sat :
  forall φ : Formula.t,
    decide φ = Sat → has_model φ.
Proof. (* 5 lines *) Qed.

Definition models_context I (ctx : context) : Prop :=
  Forall (models I) ctx.
```

```
Definition metacontext := list context.

Definition models_metacontext I := Exists (models_context I).

Definition snds {A B : Type} (xs : list (A * B)) :=
  map (@snd A B) xs.

Definition sound I (op : operator unit) : Prop := forall ctx,
  models_context I ctx →
  models_metacontext I (snds (op ctx)).

Lemma sound_bind : forall I op op',
  sound I op → sound I op' → sound I (_ ← op; op').
Proof. (* 45 lines *) Qed.

Lemma sound_split_left :
  forall I op op',
    sound I op → sound I (op ⊙ op').
Proof. (* 24 lines *) Qed.

Lemma sound_split_right :
  forall I op op',
    sound I op' → sound I (op ⊙ op').
Proof. (* 24 lines *) Qed.

Ltac destruct_if_condition H :=
  match goal with
    [ H' : context[if ?X then _ else _] |- _] =>
    match H' with
      H => destruct X
    end
  end.

Ltac destruct_if_condition_r H :=
  match goal with
    [ H' : context[if ?X then _ else _] |- _] =>
    match H' with
      H => let R := fresh in remember X as R; destruct R
    end
  end.

Ltac destruct_if_condition_in_goal :=
  match goal with
    [ |- context[if ?X then _ else _] ] => destruct X
  end.

Ltac destruct_if_condition_in_goal_r :=
  match goal with
    [ |- context[if ?X then _ else _] ] =>
    let R := fresh in remember X as R; destruct R
  end.

Lemma assumes_sound :
  forall (ϕ : Formula.t) I,
      (models I ϕ → sound I (assume ϕ))
      ∧
      (models I (fla_not ϕ) → sound I (assume_false ϕ)).
Proof. (* 177 lines *) Qed.
```

```
  Theorem decide_soundness_unsat :
    forall φ : Formula.t,
      decide φ = Unsat → ¬ has_model φ.
  Proof. (* 21 lines *) Qed.

  Theorem decide_soundness_valid :
    forall φ : Formula.t,
      decide φ = Valid → valid φ.
  Proof. (* 5 lines *) Qed.
End MakeClueful.
```

D.40 SemiconcreteExecution

```
Require Import Notations.
Require Import Basics.
Require Import EnsembleExt.
Require Import Arith.
Require Import String.
Require Import ListExt.
Require Import ListSet.
Require Import Sumbool.
Require RADefinitions.
Require RAAxioms.
Require RAOperators.

Require Nat.
Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Routine.
Require Store.
Require Heap.
Require Predicate.
Require Chunk.
Require Assertion.
Require SILPP.

Open Scope program_scope.
Open Scope bool_scope.

Module Make
  (Import RAD         : RADefinitions.DEFINITIONS)
  (Import RAA         : RAAxioms.AXIOMS RAD)
  (Import RAO         : RAOperators.AXIOMS RAD RAA).

  Module RAN := RANotations.Make(RAD).
  Include RAN.

  Module RAOPS := RAOperators.Make RAD RAA RAO.

  Import RAOPS.
  Import RAOPS.DoNotation.
  Import RAOPS.Util.

  Module RName   := Routine.DefaultName.
  Module SCStore := Store.AssocListStore Nat.
  Module SCChunk := Chunk.Make Nat.
  Module SCHeap  := Heap.Default SCChunk.

  Module StoreNotations := Store.Notations Nat SCStore.

  Import BooleanExpression.Notations.
  Import StoreNotations.
  Import SCChunk.Notations.
  Import SILPP.Notations.

  Definition zero_store := SCStore.constant_store 0.
```

```
Notation "'s_0'" := zero_store : store_scope.

Inductive semiconcrete_state : Set :=
  SemiconcreteState : SCStore.t → SCHeap.t → semiconcrete_state.

Notation "⟨ s , h ⟩" := (SemiconcreteState s h) (at level 0).

Definition store : field semiconcrete_state SCStore.t :=
  (fun σ => match σ with
              | ⟨s, _⟩ => s
            end,
   fun σ s => match σ with
                | ⟨_, h⟩ => ⟨s, h⟩
              end).

Definition heap : field semiconcrete_state SCHeap.t :=
  (fun σ => match σ with
              | ⟨_, h⟩ => h
            end,
   fun σ h => match σ with
                | ⟨s, _⟩ => ⟨s, h⟩
              end).

Definition state : field semiconcrete_state semiconcrete_state :=
  ((fun σ => σ), (fun σ σ' => σ')).

Open Scope op_scope.

Definition sc_operator :=
  operator semiconcrete_state semiconcrete_state.

Definition read_store (id : Id.t) : sc_operator nat :=
  with_current store (SCStore.lookup id).

Definition update_store (id : Id.t) (n : nat) : sc_operator unit :=
  update_current store (SCStore.bind id n).

Definition evaluate (e : Expression.t) : sc_operator nat :=
  with_current store (Expression.evaluate e ∘ flip SCStore.lookup).

Definition with_store
  {A : Set} (s : SCStore.t) (op : sc_operator A) : sc_operator A :=
  s' ← current store;
  _  ← set_current store s;
  r  ← op;
  _  ← set_current store s';
  yield r.

Definition assume_bexpr (b : BooleanExpression.t) : sc_operator unit :=
  r ← with_current store (BooleanExpression.evaluate b ∘ flip SCStore.lookup);
  assume r.

Definition assert_bexpr (b : BooleanExpression.t) : sc_operator unit :=
  r ← with_current store (BooleanExpression.evaluate b ∘ flip SCStore.lookup);
  assert r.

Definition produce_chunk (α : SCChunk.t) : sc_operator unit :=
  update_current heap (SCHeap.produce α).
```

```
Definition consume_chunk (α : SCChunk.t) : sc_operator unit :=
  r ← with_current heap (SCHeap.consume α);
  match r with
    | Some h' => set_current heap h'
    | None    => failure
  end.

Definition pick_chunk_angelically : sc_operator SCChunk.t :=
  αs ← with_current heap SCHeap.enum;
  α ← pick_angelically { α | In α αs };
  yield (proj1_sig α).

Definition find_chunk
  (pred : Predicate.t) (ℓ : nat) : sc_operator SCChunk.t :=
  α ← pick_chunk_angelically;
  match α with
    | SCChunk.Chunk p x y => assert (Predicate.beq p pred);
                             assert (beq_nat x ℓ);
                             yield (SCChunk.Chunk p x y)
  end.

Definition read_cell (ℓ : nat) : sc_operator nat :=
  α ← find_chunk Predicate.ptr ℓ;
  match α with
    | SCChunk.Chunk _ _ v => yield v
  end.

Definition write_cell (ℓ v : nat) : sc_operator unit :=
  α ← find_chunk Predicate.ptr ℓ;
  consume_chunk α;
  produce_chunk (ℓ ↦ v).

Definition clear_heap : sc_operator unit :=
  set_current heap SCHeap.empty.

Definition leak_check : sc_operator unit :=
  r ← with_current heap SCHeap.is_empty;
  assert r.

Definition alloc_set : sc_operator (list nat) :=
  let get_loc :=
    fun α => match α with
               | SCChunk.Chunk p ℓ _ =>
                 if Predicate.beq p Predicate.ptr
                   then ℓ :: nil
                   else nil
             end
    in
    h ← current heap;
    yield (concat_map get_loc h).
```

```
Fixpoint pick_demonically_n (n : nat) (A : Type) : sc_operator (list A) :=
  match n with
    | 0   => yield nil
    | S n => v ← pick_demonically A;
             vs ← pick_demonically_n n A;
             yield (v :: vs)
  end.

Definition allocate (n : nat) : sc_operator nat :=
  let allocate_at :=
    fix aux (ℓ : nat) (vs : list nat) : sc_operator unit :=
    match vs with
      | nil   => nop
      | v::vs =>
        let k := length vs in
        A ← alloc_set;
        assume (proj1_sig (bool_of_sumbool (in_dec eq_nat_dec (ℓ + k) A)));
        produce_chunk ((ℓ + k) ↦ v)%chunk;
        aux ℓ vs
    end
  in
  ns ← pick_demonically_n (S n) nat;
  match ns with
    | ℓ::vs => allocate_at ℓ vs;
               produce_chunk (mb [ℓ; n])%chunk;
               yield ℓ
    | nil   => failure
  end.

Fixpoint consume_cells (ℓ : nat) (n : nat) : sc_operator unit :=
  match n with
    | 0   => nop
    | S n => α ← find_chunk Predicate.ptr (ℓ + n);
             consume_chunk α;
             consume_cells ℓ n
  end.

Definition block_size (ℓ : nat) : sc_operator nat :=
  α ← find_chunk Predicate.mb ℓ;
  match α with
    | SCChunk.Chunk _ _ n => yield n
  end.

Fixpoint consume_assertion (a : Assertion.t) : sc_operator unit :=
  match a with
    | Assertion.bexpr b =>
      assert_bexpr b
    | Assertion.sepand a1 a2 =>
      consume_assertion a1;
      consume_assertion a2
    | Assertion.cond b a1 a2 =>
      let then_op := assume_bexpr b; consume_assertion a1 in
      let else_op := assume_bexpr (¬ b)%bexpr; consume_assertion a2 in
        op ← pick_demonically (FromList (then_op :: else_op :: nil));
        proj1_sig op
    | Assertion.pred p e x =>
      n ← evaluate e;
      α ← find_chunk p n;
      consume_chunk α;
      match α with
        | SCChunk.Chunk p n1 n2 => update_store x n2
      end
  end.

Fixpoint produce_assertion (a : Assertion.t) : sc_operator unit :=
  match a with
    | Assertion.bexpr b =>
      assume_bexpr b
    | Assertion.sepand a1 a2 =>
      produce_assertion a1;
      produce_assertion a2
    | Assertion.cond b a1 a2 =>
      let then_op  := assume_bexpr b; produce_assertion a1
      in
      let else_op := assume_bexpr (¬ b)%bexpr; produce_assertion a2
      in
        op ← pick_demonically (FromList (then_op :: else_op :: nil));
        proj1_sig op
    | Assertion.pred p e x =>
      n1 ← evaluate e;
      n2 ← pick_demonically nat;
      produce_chunk (SCChunk.Chunk p n1 n2);
      update_store x n2
  end.

Section WithProgram.
  Variable (Π : SILPP.program).

  Hypothesis (wellformed : SILPP.wellformed_program Π).

  Open Scope command_scope.

  Fixpoint semiconcrete_execution (c : SILPP.command) : sc_operator unit :=
    match c with
      | skip =>
        yield □
      | x := e =>
        v ← evaluate e;
        update_store x v
      | c; c' =>
        semiconcrete_execution c;
        semiconcrete_execution c'
      | x := malloc n =>
        ℓ ← allocate n;
        update_store x ℓ
      | free e =>
        ℓ ← evaluate e;
        n ← block_size ℓ;
        consume_cells ℓ n;
        consume_chunk (mb [ℓ; n])
      | ⟦e⟧ := e' =>
        ℓ ← evaluate e;
        v ← evaluate e';
        write_cell ℓ v
      | x := ⟦e⟧ =>
        ℓ ← evaluate e;
        v ← read_cell ℓ;
        update_store x v
      | If b Then c Else c' =>
        let then_clause :=
          assume_bexpr b;
          semiconcrete_execution c in
        let else_clause :=
          assume_bexpr (¬ b)%bexpr;
          semiconcrete_execution c' in
        let clauses := then_clause :: else_clause :: nil
        in
        op ← pick_demonically (FromList clauses);
        proj1_sig op
      | r[e] =>
        v ← evaluate e;
        rdef ← from_some (SILPP.routines Π r);
        let x := SILPP.argument rdef in
        with_store (s_0⟦x := v⟧)%store
          (consume_assertion (SILPP.precondition rdef);
           produce_assertion (SILPP.postcondition rdef))
      | open p[e; ?] =>
        ℓ ← evaluate e;
        α ← find_chunk p ℓ;
        _ ← consume_chunk α;
        preddef ← from_some (SILPP.predicates Π p);
        let x := SILPP.argument_a preddef in
        let y := SILPP.argument_b preddef in
        let body := SILPP.predicate_body preddef
        in
          match α with
            | SCChunk.Chunk _ ℓ' v' =>
            with_store (s_0⟦x := ℓ'⟧⟦y := v'⟧)%store
              (produce_assertion body)
          end
      | close p[e; e'] =>
        ℓ  ← evaluate e;
        v ← evaluate e';
        preddef ← from_some (SILPP.predicates Π p);
        let x := SILPP.argument_a preddef in
        let y := SILPP.argument_b preddef in
        let body := SILPP.predicate_body preddef in
        with_store (s_0⟦x := ℓ⟧⟦y := v⟧)%store
                   (consume_assertion body);
        produce_chunk (SCChunk.Chunk p ℓ v)
  end.
```

```
    Definition valid_routine (rdef : SILPP.routine_definition) : Prop :=
      let validation_op :=
        v ← pick_demonically nat;
        update_store (SILPP.argument rdef) v;
        produce_assertion (SILPP.precondition rdef);
        with_store (s_0⟦(SILPP.argument rdef) := v⟧)%store
                   (semiconcrete_execution (SILPP.routine_body rdef));
        consume_assertion (SILPP.postcondition rdef);
        leak_check
      in
        not (validation_op ⟨s_0, SCHeap.empty⟩ ⟺ ⊥).

    Definition valid_program : Prop :=
      forall rname : RName.t,
        match SILPP.routines Π rname with
          | Some rdef => valid_routine rdef
          | _         => True
        end.

    Close Scope command_scope.

  End WithProgram.

  Close Scope op_scope.

End Make.
```

D.41 SetHolder

```
Module Type SET.
  Parameters t : Set.
End SET.
```

D.42 SetOfSets

```
Require Import EnsembleExt.
Require Import Notations.
Require Import DependentProduct.
Require Import Relations.
Require RADefinitions.
Require RANotations.
Require RAAxioms.

Set Implicit Arguments.

Module SetOfSetDefinitions <: RADefinitions.DEFINITIONS.

  Definition ℛ (Σ : Type) := Ensemble (Ensemble Σ).

  Section StateSection.

    Variable State : Type.

    Definition single (x : State) : ℛ State :=
      Singleton (Singleton x).

    Definition models (S : Ensemble State) (R : ℛ State) : Prop :=
      exists Σ, Σ ∈ R ∧ Σ ⊆ S.

    Definition add {I : Type} (R : I → ℛ State) : ℛ State :=
      IndexedUnion R.

    Definition mul {I : Type} (R : I → ℛ State) : ℛ State :=
      fun (Σ : Ensemble State) =>
        exists f : I → Ensemble State,
          f ∈ dependent_product R ∧ Same_set _ Σ (IndexedUnion f).

    Definition top : ℛ State := Singleton ∅.

    Definition bottom : ℛ State := ∅.

    Definition implies (R R' : ℛ State) : Prop :=
      ∀ Σ, Σ ∈ R -> ∃ Σ', Σ' ∈ R' ∧ Σ' ⊆ Σ.

    Definition equiv : relation (ℛ State) :=
      fun R R' => implies R R' ∧ implies R' R.

  End StateSection.

End SetOfSetDefinitions.

Module SetOfSetAxioms <: RAAxioms.AXIOMS SetOfSetDefinitions.

  Import SetOfSetDefinitions.

  Module RAN := RANotations.Make(SetOfSetDefinitions).
  Include RAN.

  Section StateSection.

    Variable State : Type.

    Theorem top_axiom : forall (R : False → ℛ State), ⊤ ⟺ ⊗ R.
    Proof. (* 23 lines *) Qed.
    Theorem bottom_axiom :
      forall (R : False → 𝓡 State),
        ⊥ ⟺ ⊕ R.
    Proof. (* 5 lines *) Qed.

    Theorem single_axiom :
      forall (s : State) (S : Ensemble State),
        S ⊨ single s ↔ s ∈ S.
    Proof. (* 16 lines *) Qed.

    Theorem add_axiom :
      forall I (R : I → 𝓡 State) (S : Ensemble State),
        S ⊨ ⊕ R ↔ ∃ i, S ⊨ R i.
    Proof. (* 23 lines *) Qed.

    Theorem mul_axiom :
      forall I (R : I → 𝓡 State) (S : Ensemble State),
        S ⊨ ⊗ R ↔ ∀ i, S ⊨ R i.
    Proof. (* 39 lines *) Qed.

    Theorem implies_axiom :
      forall (R R' : 𝓡 State),
        R ⇛ R' ↔ forall (S : Ensemble State), S ⊨ R → S ⊨ R'.
    Proof. (* 28 lines *) Qed.

    Theorem monotonic_models_axiom :
      forall (R : 𝓡 State) S S',
        S' ⊆ S → S' ⊨ R → S ⊨ R.
    Proof. (* 7 lines *) Qed.

  End StateSection.

End SetOfSetAxioms.
```

D.43 Store

```
Require Import Notations.
Require Import ListExt.
Require Import SetHolder.
Require AssocList.

Require Import Identifier.
Require Term.

Module Type STORE (T : SET).

  Parameters
    (t               : Set)
    (lookup          : Id.t → t → T.t)
    (bind            : Id.t → T.t → t → t)
    (bound           : t → list Id.t)
    (constant_store  : T.t → t).

  Axiom lookup_bound : forall id v s, lookup id (bind id v s) = v.

  Axiom lookup_other_than_bound :
    forall id id' v s, id ≠ id' → lookup id (bind id' v s) = lookup id s.

  Axiom lookup_constant_store :
    forall id v, lookup id (constant_store v) = v.

End STORE.

Module AssocListStore (T : SET) <: STORE T.
  Module AL := AssocList.Make( Id ).

  Definition t := (T.t * AL.t T.t)%type.

  Definition constant_store (x : T.t) := (x, AL.empty T.t).

  Definition lookup (x : Id.t) (s : t) : T.t :=
    match s with
      | pair default lst => AL.lookup lst x default
    end.

  Definition bind (x : Id.t) (v : T.t) (s : t) : t :=
    match s with
      | pair default lst => (default, AL.add lst x v)
    end.

  Definition bound (s : t) : list Id.t :=
    AL.keys (snd s).

  Definition values (s : t) : list T.t :=
    AL.values (snd s).

  Theorem lookup_bound :
    forall id v s, lookup id (bind id v s) = v.
  Proof. (* 5 lines *) Qed.

  Theorem lookup_other_than_bound :
    forall id id' v s, id ≠ id' → lookup id (bind id' v s) = lookup id s.
  Proof. (* 6 lines *) Qed.
  Theorem lookup_constant_store :
    forall id v, lookup id (constant_store v) = v.
  Proof. (* 6 lines *) Qed.
End AssocListStore.

Module Notations (T : SET) (S : STORE T).
  Delimit Scope store_scope with store.
  Bind Scope store_scope with S.t.

  Notation "s ⟦ x := t ⟧" := (S.bind x t s)
    (at level 40, x at level 0, t at level 0) : store_scope.
End Notations.
```

D.44 Symbol

```
Require Import Notations.
Require Import List.
Require Import Arith.
Require Import Fresh.
Require Nat.

Module Type SYMBOL.
  Parameters
    (t      : Set)
    (eq_dec : forall x y : t, {x = y} + {x ≠ y})
    (beq    : t → t → bool)
    (fresh : forall ξs : list t, { ξ : t | ¬ In ξ ξs }).
End SYMBOL.

Module NatSymbol <: SYMBOL.
  (* Needed because of Coq8.3 limitation *)
  Inductive t' : Set :=
    Symbol : nat → t'.

  Definition t := t'.

  Definition eq_dec : forall x y : t, { x = y } + { x ≠ y } .
    intros x y; destruct x as [n]; destruct y as [n'].
    destruct (eq_nat_dec n n').
    left; subst; trivial.
    right; intro; elim n0.
    injection H; trivial.
  Defined.

  Definition beq (s s' : t) : bool.
    destruct (eq_dec s s'); [ exact true | exact false ].
  Defined.

  Definition symbol_n (s : t) : nat :=
    match s with
      | Symbol n => n
    end.

  Definition fresh (xs : list t) : { s : t | ¬ In s xs } .
    remember (Nat.fresh (map symbol_n xs)) as n.
    destruct n as [n H].
    exists (Symbol n).
    intro.
    elim H.
    clear Heqn H.
    revert n H0; induction xs; intros.
    inversion H0.
    simpl in *; destruct H0.
    subst; left; simpl; reflexivity.
    right; apply IHxs; trivial.
  Defined.
End NatSymbol.

Module Default := NatSymbol.
```

D.45 SymbolicExecution

```
Require Import Notations.
Require Import Basics.
Require Import ListExt.
Require Import ListSet.
Require Import Arith.
Require Import String.
Require Import Bool.
Require AssocList.
Require RADefinitions.
Require RAAxioms.
Require RAOperators.

Require Import Identifier.
Require Expression.
Require BooleanExpression.
Require Symbol.
Require Routine.
Require Term.
Require Formula.
Require Store.
Require Heap.
Require Predicate.
Require Chunk.
Require SILPP.
Require Assertion.
Require SMT.

Open Scope program_scope.
Open Scope bool_scope.

Module Make
  (Import RAD : RADefinitions.DEFINITIONS)
  (Import RAA : RAAxioms.AXIOMS RAD)
  (Import RAO : RAOperators.AXIOMS RAD RAA).

  Module RAN := RANotations.Make(RAD).
  Include RAN.

  Module RAOPS := RAOperators.Make RAD RAA RAO.

  Import RAOPS.
  Import RAOPS.DoNotation.
  Import RAOPS.Util.

  Module Symb   := Symbol.Default.
  Module RName  := Routine.DefaultName.
  Module SStore := Store.AssocListStore Term.
  Module SChunk := Chunk.Make Term.
  Module SHeap  := Heap.Default SChunk.

  Module StoreNotations := Store.Notations Term SStore.

  Import BooleanExpression.Notations.
  Import Term.Notations.
  Import Formula.Notations.
  Import StoreNotations.
  Import SChunk.Notations.
  Import SILPP.Notations.

  Definition zero_store := SStore.constant_store 0.

  Notation "'s_0'" := zero_store : store_scope.

  (*
     Symbolic state
  *)

  Inductive symbolic_state : Set :=
    SymbolicState : SStore.t → SHeap.t → Formula.t → symbolic_state.

  Notation "⟨ s , h , ϕ ⟩" := (SymbolicState s h ϕ) (at level 0).

  Definition state : field symbolic_state symbolic_state :=
    ((fun σ => σ), (fun σ σ' => σ')).

  Definition store : field symbolic_state SStore.t :=
    (fun σ => match σ with
              | ⟨s, _, _⟩ => s
              end,
     fun σ s => match σ with
                  | ⟨_, h, ϕ⟩ => ⟨s, h, ϕ⟩
                end).

  Definition heap : field symbolic_state SHeap.t :=
    (fun σ => match σ with
              | ⟨_, h, _⟩ => h
              end,
     fun σ h => match σ with
                  | ⟨s, _, ϕ⟩ => ⟨s, h, ϕ⟩
                end).

  Definition path_condition : field symbolic_state Formula.t :=
    (fun σ => match σ with
              | ⟨_, _, ϕ⟩ => ϕ
              end,
     fun σ ϕ => match σ with
                  | ⟨s, h, _⟩ => ⟨s, h, ϕ⟩
                end).

  Definition symbols (σ : symbolic_state) : list Symb.t :=
    match σ with
      | ⟨s, h, Φ⟩ =>
        concat_map Term.symbols (SStore.values s) ++
        concat_map Term.symbols (concat_map SChunk.args (SHeap.enum h)) ++
        Formula.symbols Φ
    end.

  Module SMTImpl := SMT.MakeClueful.

  Definition smt_result := SMT.result.

  Definition decide_sat := SMTImpl.decide.
  Definition smt_result_beq := SMT.result_beq.

  Definition Unsat := SMT.Unsat.

  Open Scope op_scope.

  Definition s_operator := operator symbolic_state symbolic_state.

  Definition read_store (id : Id.t) : s_operator Term.t :=
    with_current store (SStore.lookup id).

  Definition update_store (id : Id.t) (t : Term.t) : s_operator unit :=
    update_current store (SStore.bind id t).

  Definition evaluate (e : Expression.t) : s_operator Term.t :=
    with_current store (Term.of_expression e ∘ flip SStore.lookup).

  Definition to_formula (b : BooleanExpression.t) : s_operator Formula.t :=
    with_current store (Formula.of_boolean_expression b ∘ flip SStore.lookup).

  Definition with_store
    {A : Set} (s : SStore.t) (op : s_operator A) : s_operator A :=
    s' ← current store;
    _  ← set_current store s;
    r  ← op;
    _  ← set_current store s';
    yield r.

  Definition smt (f : Formula.t) : s_operator smt_result :=
    yield (decide_sat f).

  Definition assume_formula (ϕ : Formula.t) : s_operator unit :=
    Φ ← current path_condition;
    r ← smt (ϕ && Φ)%fla;
    assume (negb (smt_result_beq r Unsat));
    set_current path_condition (ϕ && Φ)%fla.

  Definition assume_bexpr (b : BooleanExpression.t) : s_operator unit :=
    ϕ ← with_current store
                     (Formula.of_boolean_expression b ∘ flip SStore.lookup);
    assume_formula ϕ.

  Definition assert_formula (ϕ : Formula.t) : s_operator unit :=
    Φ ← current path_condition;
    r ← smt (Formula.implies Φ ϕ);
    assert (smt_result_beq r Unsat).

  Definition assert_bexpr (b : BooleanExpression.t) : s_operator unit :=
    ϕ ← to_formula b; assert_formula ϕ.

  Definition produce_chunk (α : SChunk.t) : s_operator unit :=
    update_current heap (SHeap.produce α).

  Definition consume_chunk (α : SChunk.t) : s_operator unit :=
    r ← with_current heap (SHeap.consume α);
    match r with
      | Some h' => set_current heap h'
      | None    => failure
    end.
```

```
Definition pick_chunk_angelically : s_operator SChunk.t :=
  αs ← with_current heap SHeap.enum;
  α ← pick_angelically (FromList αs);
  yield (proj1_sig α).

Definition find_chunk (p : Predicate.t) (t : Term.t) : s_operator SChunk.t :=
  α ← pick_chunk_angelically;
  match α with
    | SChunk.Chunk p x y => assert (Predicate.beq p p);
                            assert_formula (x == t)%fla;
                            yield (SChunk.Chunk p x y)
  end.

Definition read_cell (ℓ : Term.t) : s_operator Term.t :=
  α ← find_chunk Predicate.ptr ℓ;
  match α with
    | SChunk.Chunk _ _ v => yield v
  end.

Definition write_cell (ℓ v : Term.t) : s_operator unit :=
  α ← find_chunk Predicate.ptr ℓ;
  consume_chunk α;
  produce_chunk (ℓ ↦ v).

Definition clear_heap : s_operator unit :=
  set_current heap SHeap.empty.

Definition leak_check : s_operator unit :=
  r ← with_current heap SHeap.is_empty; assert r.

Definition fresh_symbol (exclude : list Symb.t) : s_operator Symb.t :=
  let ξ := (proj1_sig (Symb.fresh exclude))
  in
  assume_formula (Formula.fla_eq (Term.term_sym ξ) (Term.term_sym ξ));
  yield ξ.

Definition fresh_symbol_n (n : nat) : s_operator (list Symb.t) :=
  let aux  :=
    fix rec (n : nat) (exclude : list Symb.t) : s_operator (list Symb.t) :=
    match n with
      | 0   => yield nil
      | S n => ξ ← fresh_symbol exclude;
               ξs ← rec n (ξ :: exclude);
               yield (ξ :: ξs)
    end
    in
    σ ← current state;
    let exclude := symbols σ
    in
    aux n exclude.

Definition fresh_symbol_1 : s_operator Symb.t :=
  ξs ← fresh_symbol_n 1;
  match ξs with
    | ξ :: nil => yield ξ
    | _        => failure
  end.
```

```
Definition allocate (n : nat) : s_operator Term.t :=
  let allocate_at :=
    fix aux (ℓ : Symb.t) (vs : list Symb.t) : s_operator unit :=
    match vs with
      | nil   => nop
      | v::vs =>
        let k := List.length vs in
        let α := ((Term.term_sym ℓ + k)%term ↦ (Term.term_sym v))%chunk in
        produce_chunk α;
        aux ℓ vs
    end
  in
  ξs ← fresh_symbol_n (S n);
  match ξs with
    | ℓ::vs => allocate_at ℓ vs;
               produce_chunk (mb [(Term.term_sym ℓ); n])%chunk;
               yield (Term.term_sym ℓ)
    | nil   => failure
  end.

Fixpoint consume_cells (ℓ : Term.t) (n : nat) : s_operator unit :=
  match n with
    | O   => nop
    | S n => α ← find_chunk Predicate.ptr (ℓ + n);
             consume_chunk α;
             consume_cells ℓ n
  end.

Definition block_size (ℓ : Term.t) : s_operator nat :=
  α ← find_chunk Predicate.mb ℓ;
  match α with
    | SChunk.Chunk _ _ (Term.term_lit n) => yield n
    | _                                  => failure
  end.

Fixpoint consume_assertion (a : Assertion.t) : s_operator unit :=
  match a with
    | Assertion.bexpr b =>
      assert_bexpr b
    | Assertion.sepand a1 a2 =>
      consume_assertion a1;
      consume_assertion a2
    | Assertion.cond b a1 a2 =>
      let then_op := assume_bexpr b; consume_assertion a1 in
      let else_op := assume_bexpr (¬ b)%bexpr; consume_assertion a2 in
        op ← pick_demonically (FromList (then_op :: else_op :: nil));
        proj1_sig op
    | Assertion.pred p e x =>
      t ← evaluate e;
      α ← find_chunk p t;
      consume_chunk α;
      match α with
        | SChunk.Chunk p t1 t2 => update_store x t2
      end
  end.
```

```
Fixpoint produce_assertion (a : Assertion.t) : s_operator unit :=
  match a with
    | Assertion.bexpr b =>
      assume_bexpr b
    | Assertion.sepand a1 a2 =>
      produce_assertion a1;
      produce_assertion a2
    | Assertion.cond b a1 a2 =>
      let then_op  := assume_bexpr b; produce_assertion a1 in
      let else_op := assume_bexpr (¬ b)%bexpr; produce_assertion a2
      in
        op ← pick_demonically (FromList (then_op :: else_op :: nil));
        proj1_sig op
    | Assertion.pred p e x =>
      t1 ← evaluate e;
      ξ ← fresh_symbol_1;
      let t2 := Term.term_sym ξ in
      produce_chunk (SChunk.Chunk p t1 t2);
      update_store x t2
  end.

Section WithProgram.

  Variable (Π : SILPP.program).

  Hypothesis (wellformed : SILPP.wellformed_program Π).

  Open Scope command_scope.

  Fixpoint symbolic_execution (c : SILPP.command) : s_operator unit :=
    match c with
      | skip =>
        yield □
      | x := e =>
        v ← evaluate e;
        update_store x v
      | c; c' =>
        symbolic_execution c;
        symbolic_execution c'
      | x := malloc n =>
        ℓ ← allocate n;
        update_store x ℓ
      | free e =>
        ℓ ← evaluate e;
        n ← block_size ℓ;
        consume_cells ℓ n;
        consume_chunk (mb [ℓ; n])
      | ⟦e⟧ := e' =>
        ℓ ← evaluate e;
        v ← evaluate e';
        write_cell ℓ v
      | x := ⟦e⟧ =>
        ℓ ← evaluate e;
        v ← read_cell ℓ;
        update_store x v
      | If b Then c Else c' =>
        let then_clause :=
          assume_bexpr b; symbolic_execution c in
        let else_clause :=
          assume_bexpr (¬ b)%bexpr; symbolic_execution c' in
        let clauses := then_clause :: else_clause :: nil in
        op ← pick_demonically (FromList clauses);
        proj1_sig op
      | open p[e; ?] =>
        ℓ ← evaluate e;
        α ← find_chunk p ℓ;
        _ ← consume_chunk α;
        preddef ← from_some (SILPP.predicates Π p);
        let x := SILPP.argument_a preddef in
        let y := SILPP.argument_b preddef in
        let body := SILPP.predicate_body preddef in
          match α with
            | SChunk.Chunk _ ℓ' v' =>
              with_store (s_0⟦x := ℓ'⟧⟦y := v'⟧)%store
                         (produce_assertion body)
          end
      | close p[e; e'] =>
        ℓ ← evaluate e; v ← evaluate e';
        preddef ← from_some (SILPP.predicates Π p);
        let x := SILPP.argument_a preddef in
        let y := SILPP.argument_b preddef in
        let body := SILPP.predicate_body preddef in
        with_store (s_0⟦x := ℓ⟧⟦y := v⟧)%store
                   (consume_assertion body);
        produce_chunk (SChunk.Chunk p ℓ v)
      | r[e] =>
        rdef ← from_some (SILPP.routines Π r);
        let x := SILPP.argument rdef in
        t ← evaluate e;
        with_store (s_0⟦x := t⟧)%store
          (consume_assertion (SILPP.precondition rdef);
           produce_assertion (SILPP.postcondition rdef))
    end.

  Definition valid_routine (rdef : SILPP.routine_definition) : Prop :=
    let validation_op :=
      ξ ← fresh_symbol_1;
      let v := Term.term_sym ξ in
      update_store (SILPP.argument rdef) v;
      produce_assertion (SILPP.precondition rdef);
      with_store (s_0⟦(SILPP.argument rdef) := v⟧)%store
                 (symbolic_execution (SILPP.routine_body rdef));
      consume_assertion (SILPP.postcondition rdef);
      leak_check
    in
    let pc := Formula.fla_eq (Term.term_lit 0) (Term.term_lit 0) in
    let σ := ⟨s_0, SHeap.empty, pc ⟩ in
      not (validation_op σ ⟺ ⊥).
    Definition valid_program : Prop :=
      forall rname : RName.t,
        match SILPP.routines Π rname with
          | Some rdef => valid_routine rdef
          | _         => True
        end.

    Close Scope command_scope.

  End WithProgram.

  Close Scope op_scope.

End Make.
```

D.46 Term

```
Require Import Notations.
Require Import List.
Require Import Arith.
Require Import Identifier.
Require Symbol.
Require Expression.

Module Symb := Symbol.Default.

Inductive t' : Set :=
  | term_lit : nat     → t'
  | term_sym : Symb.t  → t'
  | term_add : t'      → t' → t'
  | term_sub : t'      → t' → t'
  | term_mul : t'      → t' → t'.

Definition t := t'.

Definition add (t1 t2 : t) :=
  match t1, t2 with
    | term_lit 0, _ => t2
    | _, term_lit 0 => t1
    | _, _          => term_add t1 t2
  end.

Definition eq_dec : forall t1 t2 : t, {t1 = t2} + {t1 ≠ t2} .
  induction t1; destruct t2; try (right; discriminate; fail).

  (* term_lit *)
  destruct (eq_nat_dec n n0).
  left; subst; reflexivity.
  right; intro; elim n1.
  injection H; trivial.

  (* term_sym *)
  rename t0 into ξ1; rename t1 into ξ2.
  destruct (Symb.eq_dec ξ1 ξ2).
  left; subst; reflexivity.
  right; intro; elim n.
  injection H.
  trivial.

  (* term_add *)
  destruct (IHt1_1 t2_1); destruct (IHt1_2 t2_2); clear IHt1_1 IHt1_2; subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.

  (* term_sub *)
  destruct (IHt1_1 t2_1); destruct (IHt1_2 t2_2); clear IHt1_1 IHt1_2; subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.

  (* term_mul *)
  destruct (IHt1_1 t2_1); destruct (IHt1_2 t2_2); clear IHt1_1 IHt1_2; subst;
    try (left; reflexivity; fail); right; intro; elim n; injection H; trivial.
Defined.
```

```
Definition beq (t1 t2 : t) : bool.
  destruct (eq_dec t1 t2); [ exact true | exact false ].
Defined.

Fixpoint symbols (term : t) : list Symb.t :=
  match term with
    | term_lit _    => nil
    | term_sym ξ    => ξ :: nil
    | term_add t1 t2 => symbols t1 ++ symbols t2
    | term_sub t1 t2 => symbols t1 ++ symbols t2
    | term_mul t1 t2 => symbols t1 ++ symbols t2
  end.

Fixpoint of_expression (e : Expression.t) (s : Id.t → t) : t :=
  match e with
    | Expression.lit n     => term_lit n
    | Expression.var x     => s x
    | Expression.add e e'  => term_add (of_expression e s) (of_expression e' s)
    | Expression.min e e'  => term_sub (of_expression e s) (of_expression e' s)
    | Expression.mul e e'  => term_mul (of_expression e s) (of_expression e' s)
  end.

Definition nat_to_term (n : nat) : t := term_lit n.
Definition sym_to_term (x : Symb.t) : t := term_sym x.

Fixpoint evaluate (term : t) (s : Symb.t → nat) : nat :=
  match term with
    | term_lit n       => n
    | term_sym a       => s a
    | term_add t t'    => evaluate t s + evaluate t' s
    | term_sub t t'    => evaluate t s - evaluate t' s
    | term_mul t t'    => evaluate t s * evaluate t' s
  end.

Fixpoint metric (term : t) : nat :=
  match term with
    | term_lit _ => 1
    | term_sym _ => 1
    | term_add t t' => S (metric t + metric t')
    | term_sub t t' => S (metric t + metric t')
    | term_mul t t' => S (metric t + metric t')
  end.

Module Notations.
  Coercion term_lit    : nat    >-> t'.
  Coercion term_sym    : Symb.t >-> t'.
  Coercion nat_to_term : nat    >-> t.
  Coercion sym_to_term : Symb.t >-> t.

  Delimit Scope term_scope with term.
  Bind Scope term_scope with t.

  Infix "+" := add : term_scope.
  Infix "-" := term_sub : term_scope.
  Infix "*" := term_mul : term_scope.
End Notations.
```

D.47 VCG

```
Require Import Notations.
Require Import Classical.
Require Import FunctionalExtensionality.

(*
   Language definition
*)
Parameters
  (identifier : Set)
  (value      : Set)
  (t          : value).

Definition store := identifier → value.

Definition expression := store → value.

Axiom identifier_eq_dec :
  forall x y : identifier, {x = y} + {x ≠ y} .

Axiom value_eq_dec :
  forall v v' : value, {v = v'} + {v ≠ v'} .

Inductive command : Set :=
| cAssert : expression → command
| cAssume : expression → command
| cHavoc  : identifier → command
| cAssign : identifier → expression → command
| cChoice : command → command → command
| cSeq    : command → command → command
| cSkip   : command.

Inductive state : Set :=
| in_progress : command → store → state
| failure     : store → state.

Notation "⟨ c ; s ⟩" := (in_progress c s).

Definition fupd {A B}
  (eq_dec : forall x x' : A, {x = x'}+{x ≠ x'})
  (f : A → B) (x : A) (y : B) : A → B :=
  fun x' => if eq_dec x x' then y else f x'.

Definition rebind := @fupd identifier value identifier_eq_dec.

Reserved Notation "x --> y" (at level 50).

Inductive step : state → state → Prop :=
| stepAssertTrue : forall (e : expression) (μ : store),
  e μ = t → ⟨cAssert e; μ⟩ --> ⟨cSkip; μ⟩
| stepAssertFalse : forall (e : expression) (μ : store),
  e μ <> t → ⟨cAssert e; μ⟩ --> (failure μ)
| stepAssumeTrue : forall (e : expression) (μ : store),
  e μ = t → ⟨cAssume e; μ⟩ --> ⟨cSkip; μ⟩
| stepHavoc : forall (x : identifier) (v : value) (μ : store),
  ⟨cHavoc x; μ⟩ --> ⟨cSkip; rebind μ x v⟩
```

```
| stepAssign : forall (x : identifier) (e : expression) (μ : store),
  ⟨cAssign x e; μ⟩ --> ⟨cSkip; rebind μ x (e μ)⟩
| stepChoiceLeft : forall (c c' : command) (μ : store),
  ⟨cChoice c c'; μ⟩ --> ⟨c; μ⟩
| stepChoiceRight : forall (c c' : command) (μ : store),
  ⟨cChoice c c'; μ⟩ --> ⟨c'; μ⟩
| stepSeq : forall (c1 c1' c2 : command) (μ μ' : store),
  ⟨c1; μ⟩ --> ⟨c1'; μ'⟩ →
  ⟨cSeq c1 c2; μ⟩ --> ⟨cSeq c1' c2; μ'⟩
| stepSeqSkip : forall (c : command) (μ : store),
  ⟨cSeq cSkip c; μ⟩ --> ⟨c; μ⟩
| stepSeqFail : forall (c1 c2 : command) (μ μ' : store),
  ⟨c1; μ⟩ --> (failure μ') →
  ⟨cSeq c1 c2; μ⟩ --> (failure μ')
where "x --> y" := (step x y).

Reserved Notation "x -->* y" (at level 50).

Inductive steps : state → state → Prop :=
| stepsReflexivity : forall σ : state, σ -->* σ
| stepsStep        : forall σ σ' σ" : state, σ --> σ' → σ' -->* σ" → σ -->* σ"
where "x -->* y" := (steps x y).

Definition reducible (σ : state) := exists σ' : state, σ --> σ'.

Definition irreducible (σ : state) := ¬ reducible σ.

Lemma irreducible_steps :
  forall σ σ' : state,
    irreducible σ → σ -->* σ' → σ = σ'.
Proof. (* 5 lines *) Qed.

Fixpoint final_state (σ : state) : Prop :=
  match σ with
    | failure _    => True
    | ⟨cSkip; _⟩  => True
    | _            => False
  end.

Fixpoint leftmost_assume (c : command) : option expression :=
  match c with
    | cSeq c _  => leftmost_assume c
    | cAssume e => Some e
    | _         => None
  end.

Definition stuck_assume (σ : state) : Prop :=
  match σ with
    | ⟨c; μ⟩ => exists e, leftmost_assume c = Some e ∧ e μ ≠ t
    | _      => False
  end.

Theorem irreducible_stuck_assume :
  forall σ : state,
    stuck_assume σ → irreducible σ.
Proof. (* 14 lines *) Qed.
```

```
Inductive is_irred : state → Set :=
| is_irred_assume  : forall σ : state, stuck_assume σ → is_irred σ
| is_irred_failure : forall μ : store, is_irred (failure μ)
| is_irred_skip    : forall μ : store, is_irred ⟨cSkip; μ⟩.

Theorem is_irred_impl_irreducible :
  forall σ : state, is_irred σ → irreducible σ.
Proof. (* 4 lines *) Qed.

Theorem irreducible_impl_is_irred :
  forall σ : state, irreducible σ → is_irred σ.
Proof. (* 54 lines *) Qed.

Theorem irreducible_states :
  forall σ : state,
    irreducible σ ↔ stuck_assume σ ∨ exists μ, σ = failure μ ∨ σ = ⟨cSkip; μ⟩.
Proof. (* 10 lines *) Qed.

Definition bigstep (σ σ' : state) : Prop :=
  steps σ σ' ∧ final_state σ'.

Notation "x -->| y" := (bigstep x y) (at level 50).

Definition store_predicate := store → Prop.

Lemma lift_step :
  forall σ σ' : state, σ --> σ' → σ -->* σ'.
Proof. (* 2 lines *) Qed.

Lemma irreducible_final_state :
  forall (σ : state), final_state σ → irreducible σ.
Proof. (* 3 lines *) Qed.

Lemma final_state_steps :
  forall (σ σ' : state), final_state σ → σ -->* σ' → σ = σ'.
Proof. (* 5 lines *) Qed.

Ltac assert_final_state_as H σ :=
  assert (H : final_state σ);
    [ try (trivial; compute; trivial; fail)
    | idtac ].

Ltac assert_final_state σ :=
  let id := fresh in assert_final_state_as id σ.

Ltac to_bigstep H :=
  match goal with
    | [ H' : ?X -->* ?Y |- _ ] =>
      match H with
        | H' => let id := fresh in
                let id' := fresh
                in
                  assert_final_state_as id Y;
                  assert (id' : X -->| Y); [ red; split; trivial; fail | idtac ];
                  clear id H';
                  rename id' into H'
                end
  end.
```

```
Ltac reflexive_steps H :=
  match goal with
    | [ H' : ?X -->* ?Y |- _ ] =>
      match H with
        | H' => let id := fresh in
                  assert_final_state_as id X;
                  apply (final_state_steps _ _ id) in H;
                  clear id
      end
  end.

Lemma skip_steps_skip : forall σ μ, ⟨cSkip; μ⟩ -->* σ → σ = ⟨cSkip; μ⟩.
Proof. (* 3 lines *) Qed.

Lemma failure_steps_failure : forall σ μ, failure μ -->* σ → σ = failure μ.
Proof. (* 3 lines *) Qed.

Ltac remember_states H x y :=
  match goal with
    | [ H' : ?X -->  ?Y |- _ ] => match H with
                                    H' => remember X as x; remember Y as y
                                  end
    | [ H' : ?X -->* ?Y |- _ ] => match H with
                                    H' => remember X as x; remember Y as y
                                  end
    | [ H' : ?X -->| ?Y |- _ ] => match H with
                                    H' => remember X as x; remember Y as y
                                  end
  end.

Lemma seq_skips : forall (c1 c2 : command) (μ μ' : store),
  ⟨cSeq c1 c2; μ⟩ -->* ⟨cSkip; μ'⟩ →
  exists μ" : store,
    ⟨c1; μ⟩ -->* ⟨cSkip; μ"⟩ ∧
    ⟨c2; μ"⟩ -->* ⟨cSkip; μ'⟩.
Proof. (* 14 lines *) Qed.

Lemma seq_fails : forall (c1 c2 : command) (μ μ' : store),
  ⟨cSeq c1 c2; μ⟩ -->* (failure μ') →
  ⟨c1; μ⟩ -->* (failure μ') ∨
  exists μ", ⟨c1; μ⟩ -->* ⟨cSkip; μ"⟩ ∧
             ⟨c2; μ"⟩ -->* failure μ'.
Proof. (* 20 lines *) Qed.

Definition fails (σ : state) := exists μ, σ -->* failure μ.

Definition succeeds (σ : state) := ¬ fails σ.

Ltac step s :=
  match goal with
    | [ |- ?X -->* ?Y ] => apply (stepsStep X s Y)
  end.

Ltac single_step :=
  match goal with
    | [ |- ?X -->* ?Y ] => refine (stepsStep X Y Y _ (stepsReflexivity _))
  end.
```

```
Ltac destruct_conjunctions :=
  match goal with
    | [ H : _ ∧ _ |- _ ] => destruct H; destruct_conjunctions
    | _ => idtac
  end.

Theorem append_steps :
  forall (σ σ' σ" : state),
    σ -->* σ' → σ' -->* σ" → σ -->* σ".
Proof. (* 7 lines *) Qed.

Theorem steps_seq :
  forall (c1 c1' c2 : command) (μ μ' : store),
    ⟨c1; μ⟩ -->* ⟨c1'; μ'⟩ → ⟨cSeq c1 c2; μ⟩ -->* ⟨cSeq c1' c2; μ'⟩.
Proof. (* 12 lines *) Qed.

Theorem steps_seq_skip :
  forall (c1 c2 : command) (μ μ' : store),
    ⟨c1; μ⟩ -->* ⟨cSkip; μ'⟩ → ⟨cSeq c1 c2; μ⟩ -->* ⟨c2; μ'⟩.
Proof. (* 5 lines *) Qed.

Ltac goal_big_to_single :=
  red; split;
    [ single_step | compute; trivial; fail ].

Theorem reducible_dec :
  forall (σ : state),
    { reducible σ } + { irreducible σ } .
Proof. (* 44 lines *) Qed.

Theorem final_dec : forall σ : state, { final_state σ } + { ¬ final_state σ } .
Proof. (* 3 lines *) Qed.

(*
   Weakest preconditions
*)

Fixpoint weakest_precondition
  (c : command)
  (Q : store_predicate) : store_predicate :=
  match c with
    | cAssert e    =>
      (fun μ => (e μ = t ∧ Q μ))
    | cAssume e    =>
      (fun μ => (e μ = t → Q μ))
    | cHavoc x     =>
      (fun μ => forall v, Q (rebind μ x v))
    | cSkip        =>
      Q
    | cChoice c c' =>
      (fun μ => weakest_precondition c Q μ ∧ weakest_precondition c' Q μ)
    | cSeq c c'    =>
      (fun μ => weakest_precondition c (weakest_precondition c' Q) μ)
    | cAssign x e  =>
      (fun μ => Q (rebind μ x (e μ)))
  end.
```

```
Lemma wp_skip :
  forall (c : command) (σ : state) (μ : store) (Q : store_predicate),
    weakest_precondition c Q μ → ⟨c; μ⟩ -->| σ →
      exists μ', σ = ⟨cSkip; μ'⟩ ∧ Q μ'.
Proof. (* 97 lines *) Qed.

Definition nofail (c : command) := weakest_precondition c (fun _ => True).

Theorem wp_nofail : forall (c : command) (μ : store),
  weakest_precondition c (fun _ => True) μ → succeeds ⟨c; μ⟩.
Proof. (* 7 lines *) Qed.

Theorem wp_postcondition :
  forall (c : command) (μ μ' : store) (Q : store_predicate),
    weakest_precondition c Q μ → ⟨c; μ⟩ -->* ⟨cSkip; μ'⟩ → Q μ'.
Proof. (* 5 lines *) Qed.

(*
   Strongest postcondition
*)

Fixpoint strongest_postcondition
  (c : command)
  (P : store_predicate) : store_predicate :=
  match c with
    | cAssert e    =>
      (fun μ => e μ = t ∧ P μ)
    | cAssume e    =>
      (fun μ => e μ = t ∧ P μ)
    | cHavoc x     =>
      (fun μ => exists v : value, P (rebind μ x v))
    | cSkip        =>
      P
    | cChoice c c' =>
      (fun μ => strongest_postcondition c P μ ∨ strongest_postcondition c' P μ)
    | cSeq c c'    =>
      (fun μ => strongest_postcondition c' (strongest_postcondition c P) μ)
    | cAssign x e  =>
      (fun μ => exists v, P (rebind μ x v) ∧ μ x = e (rebind μ x v))
  end.

Theorem sp_soundness :
  forall (c : command) (μ μ' : store) (P : store_predicate),
    P μ → ⟨c; μ⟩ -->* ⟨cSkip; μ'⟩ → strongest_postcondition c P μ'.
Proof. (* 86 lines *) Qed.

Theorem sp_strongest_aux : forall (c : command) (μ' : store) (P :
store_predicate),
  strongest_postcondition c P μ' → exists μ : store, P μ ∧ ⟨c; μ⟩ -->* ⟨cSkip;
μ'⟩.
Proof. (* 57 lines *) Qed.

Theorem sp_strongest : forall (c : command) (P Q : store_predicate),
  (forall μ μ', P μ → ⟨c; μ⟩ -->* ⟨cSkip; μ'⟩ → Q μ') →
  (forall μ', strongest_postcondition c P μ' → Q μ').
Proof. (* 3 lines *) Qed.
```

```
(*
   Weakest Liberal Preconditions
*)

Fixpoint weakest_liberal_precondition (c : command)
                                       (Q : store_predicate) : store_predicate :=
  match c with
    | cAssert e    => (fun μ => (e μ = t → Q μ))
    | cAssume e    => (fun μ => (e μ = t → Q μ))
    | cHavoc x     => (fun μ => forall v, Q (rebind μ x v))
    | cSkip        => Q
    | cChoice c c' => (fun μ => weakest_liberal_precondition c Q μ ∧
                                weakest_liberal_precondition c' Q μ)
    | cSeq c c'    => (fun μ => weakest_liberal_precondition c
                                  (weakest_liberal_precondition c' Q) μ)
    | cAssign x e  => (fun μ => Q (rebind μ x (e μ)))
  end.

Theorem wlp_soundness :
  forall (c : command) (μ μ' : store) (Q : store_predicate),
    weakest_liberal_precondition c Q μ → ⟨c; μ⟩ -->* ⟨cSkip; μ'⟩ → Q μ'.
Proof. (* 42 lines *) Qed.

Theorem wlp_sp :
  forall (c : command) (μ : store) (P Q : store_predicate),
    (forall μ, strongest_postcondition c P μ → Q μ) →
      P μ → weakest_liberal_precondition c Q μ.
Proof. (* 64 lines *) Qed.

Theorem sp_wlp :
  forall (c : command) (μ : store) (P Q : store_predicate),
    (forall μ, P μ → weakest_liberal_precondition c Q μ) →
      strongest_postcondition c P μ → Q μ.
Proof. (* 47 lines *) Qed.

Theorem wlp_sp' :
  forall (c : command) (P : store_predicate) (μ : store),
    P μ -> weakest_liberal_precondition c (strongest_postcondition c P) μ.
Proof. (* 3 lines *) Qed.

Theorem sp_wlp' :
  forall (c : command) (Q : store_predicate) (μ : store),
    strongest_postcondition c (weakest_liberal_precondition c Q) μ -> Q μ.
Proof. (* 3 lines *) Qed.

Theorem wlp_weakest :
  forall (c : command) (P Q : store_predicate),
    (forall μ μ', P μ → ⟨c; μ⟩ -->* ⟨cSkip; μ'⟩ → Q μ') →
    (forall μ, P μ → weakest_liberal_precondition c Q μ).
Proof. (* 4 lines *) Qed.

Theorem wlp_monotonic :
  forall (c : command) (Q Q' : store_predicate) (μ : store),
    (forall μ, Q μ → Q' μ) → weakest_liberal_precondition c Q μ →
      weakest_liberal_precondition c Q' μ.
Proof. (* 9 lines *) Qed.
```

```
Theorem wp_monotonic :
  forall (c : command) (Q Q' : store_predicate) (μ : store),
    (forall μ, Q μ → Q' μ) → weakest_precondition c Q μ →
    weakest_precondition c Q' μ.
Proof. (* 12 lines *) Qed.

Theorem wp_impl_wlp :
  forall (c : command) (Q : store_predicate) (μ : store),
    weakest_precondition c Q μ → weakest_liberal_precondition c Q μ.
Proof. (* 12 lines *) Qed.

Theorem wp_impl_nofail :
  forall (c : command) (Q : store_predicate) (μ : store),
    weakest_precondition c Q μ → nofail c μ.
Proof. (* 3 lines *) Qed.

Theorem wlp_conj :
  forall (c : command) (Q Q' : store_predicate) (μ : store),
    weakest_liberal_precondition c Q μ →
    weakest_liberal_precondition c Q' μ →
    weakest_liberal_precondition c (fun μ => Q μ ∧ Q' μ) μ.
Proof. (* 8 lines *) Qed.

Theorem wp_as_wlp_and_nf :
  forall (c : command) (Q : store_predicate) (μ : store),
    weakest_precondition c Q μ ↔
      weakest_liberal_precondition c Q μ ∧ nofail c μ.
Proof. (* 22 lines *) Qed.
```

D.48 WeakestPreconditions

```
Require Import EnsembleExt.
Require Import Notations.
Require Import Classical.
Require Import Relations.
Require Import Setoid.
Require Import Basics.
Require Import Misc.
Require Import ClassicalChoice.
Require Import DependentProduct.
Require RADefinitions.
Require RANotations.
Require RAAxioms.
Require RATheorems.
Require RAOperators.
Require RAOperatorTheorems.

Set Implicit Arguments.

Module WeakestPreconditionDefinitions
  <: RADefinitions.DEFINITIONS.

  Definition 𝓡_raw (Σ : Type) := Ensemble (Ensemble Σ).

  Definition 𝓡 (Σ : Type) := { R : 𝓡_raw Σ | closed R } .

  Definition unpack {Σ} (R : 𝓡 Σ) : 𝓡_raw Σ :=
    proj1_sig R.

  Definition f_unpack {Σ I} (R : I → 𝓡 Σ) : I → 𝓡_raw Σ :=
    compose (@unpack Σ) R.

  Theorem closed_unpack :
    forall {Σ} (R : 𝓡 Σ),
      closed (unpack R).
  Proof. (* 2 lines *) Qed.

  Theorem closed_f_unpack :
    forall {Σ I} (R : I → 𝓡 Σ) i,
      closed ((f_unpack R) i).
  Proof. (* 3 lines *) Qed.

  Section StateSection.

    Variable State : Type.

    Definition single_raw (x : State) : 𝓡_raw State :=
      fun Σ => x ∈ Σ.

    Definition single (x : State) : 𝓡 State.
    exists (single_raw x).
    compute; intros.
    apply (H0 x H).
    Defined.

    Definition models_raw (S : Ensemble State) (R : 𝓡_raw State) : Prop :=
      exists Σ, Σ ∈ R ∧ Same_set _ Σ S.
    Definition models (S : Ensemble State) (R : 𝓡 State) : Prop :=
      models_raw S (unpack R).

    Definition add_raw {I} (R : I → 𝓡_raw State) : 𝓡_raw State :=
      IndexedUnion R.

    Definition add {I} (R : I → 𝓡 State) : 𝓡 State.
    exists (add_raw (f_unpack R)).
    apply closed_IndexedUnion.
    apply closed_f_unpack.
    Defined.

    Definition mul_raw {I} (R : I → 𝓡_raw State) : 𝓡_raw State :=
      IndexedIntersection R.

    Definition mul {I} (R : I → 𝓡 State) : 𝓡 State.
    exists (mul_raw (f_unpack R)).
    apply closed_IndexedIntersection.
    apply closed_f_unpack.
    Defined.

    Definition top_raw : 𝓡_raw State :=
      fun _ => True.

    Definition top : 𝓡 State.
    exists (top_raw).
    compute; trivial.
    Defined.

    Definition bottom_raw : 𝓡_raw State := ∅.

    Definition bottom : 𝓡 State.
    exists bottom_raw.
    apply closed_empty_set.
    Defined.

    Definition implies_raw (R R' : 𝓡_raw State) : Prop :=
      R ⊆ R'.

    Definition implies (R R' : 𝓡 State) : Prop :=
      implies_raw (unpack R) (unpack R').

    Definition equiv_raw : relation (𝓡_raw State) :=
      fun R R' => implies_raw R R' ∧ implies_raw R' R.

    Definition equiv : relation (𝓡 State) :=
      fun R R' => implies R R' ∧ implies R' R.

  End StateSection.

End WeakestPreconditionDefinitions.

Module WeakestPreconditionAxioms
        <: RAAxioms.AXIOMS WeakestPreconditionDefinitions.

  Import WeakestPreconditionDefinitions.

  Module RAN :=
    RANotations.Make( WeakestPreconditionDefinitions ).
  Include RAN.
  Section StateSection.

    Variable State : Type.

    Theorem top_axiom : forall (R : False → 𝓡 State),
        ⊤ ⟸⟹ ⊗ R.
    Proof. (* 2 lines *) Qed.

    Theorem bottom_axiom : forall (R : False → 𝓡 State),
        ⊥ ⟸⟹ ⊕ R.
    Proof. (* 4 lines *) Qed.

    Theorem single_axiom :
      forall (s : State) (S : Ensemble State),
        S ⊨ single s ↔ s ∈ S.
    Proof. (* 7 lines *) Qed.

    Theorem add_axiom :
      forall I (R : I → 𝓡 State) (S : Ensemble State),
        S ⊨ ⊕ R ↔ ∃ i, S ⊨ R i.
    Proof. (* 11 lines *) Qed.

    Theorem mul_axiom :
      forall I (R : I → 𝓡 State) (S : Ensemble State),
        S ⊨ ⊗ R ↔ ∀ i, S ⊨ R i.
    Proof. (* 16 lines *) Qed.

    Theorem implies_axiom :
      forall (R R' : 𝓡 State),
        R ⇛ R' ↔ forall (S : Ensemble State), S ⊨ R → S ⊨ R'.
    Proof. (* 10 lines *) Qed.

    Theorem monotonic_models_axiom :
      forall (R : 𝓡 State) S S',
        S' ⊆ S → S' ⊨ R → S ⊨ R.
    Proof. (* 7 lines *) Qed.

  End StateSection.

End WeakestPreconditionAxioms.

Module WeakestPreconditionOperators.

  Import WeakestPreconditionDefinitions WeakestPreconditionAxioms.

  Module RAN :=
    RANotations.Make ( WeakestPreconditionDefinitions ).
  Include RAN.

  Module RAT :=
    RATheorems.Make
      ( WeakestPreconditionDefinitions )
      ( WeakestPreconditionAxioms ).
  Import RAT.

  Definition bind_raw {Σ Σ' Σ"}
                      (f : Σ → 𝓡_raw Σ')
                      (g : Σ' → 𝓡_raw Σ") : Σ → 𝓡_raw Σ" :=
    flip (compose (flip f) (flip g)).
```

```
Definition bind {Σ Σ' Σ"} (f : Σ → 𝓡 Σ') (g : Σ' → 𝓡 Σ") : Σ → 𝓡 Σ".
  remember (f_unpack f) as f'; remember (f_unpack g) as g'.
  assert (forall x, closed (f' x)).
  intro; subst; apply closed_f_unpack.
  assert (forall x, closed (g' x)).
  intro; subst; apply closed_f_unpack.
  intro x.
  exists ((bind_raw f' g') x).
  unfold bind_raw.
  generalize H H0; clear; intros.
  rename x into S.
  compute [ flip compose closed In ]; intros.
  specialize (H S); red in H.
  apply (H _ _ H1).
  compute [ Included In ]; intros.
  specialize (H0 x); red in H0.
  apply (H0 _ _ H3).
  exact H2.
Defined.

Theorem f_unpack_bind :
  forall {Σ Σ' Σ"} (f : Σ → 𝓡 Σ') (g : Σ' → 𝓡 Σ"),
    f_unpack (bind f g) = bind_raw (f_unpack f) (f_unpack g).
Proof. (* 1 lines *) Qed.

Theorem unpack_bind :
  forall {Σ Σ' Σ"} (f : Σ → 𝓡 Σ') (g : Σ' → 𝓡 Σ") x,
    unpack (bind f g x) = bind_raw (f_unpack f) (f_unpack g) x.
Proof. (* 1 lines *) Qed.

Theorem bind_associative :
  forall {Σ Σ' Σ" Σ"'}
         (f : Σ → 𝓡 Σ')
         (g : Σ' → 𝓡 Σ")
         (h : Σ" → 𝓡 Σ"'),
    f_equiv (bind (bind f g) h) (bind f (bind g h)).
Proof. (* 9 lines *) Qed.

Definition lift_raw {Σ Σ'} (f : Σ → 𝓡_raw Σ') : 𝓡_raw Σ → 𝓡_raw Σ' :=
  bind_raw id f.

Definition lift {Σ Σ'} (f : Σ → 𝓡 Σ') : 𝓡 Σ → 𝓡 Σ' :=
  bind id f.

Section StateSection.

  Variables (Σ Σ' : Type).

  Theorem monotonic_lift_axiom :
    forall (f g : Σ → 𝓡 Σ') (R R' : 𝓡 Σ),
      f_implies f g → R ⇛ R' → lift f R ⇛ lift g R'.
  Proof. (* 18 lines *) Qed.

  Theorem lift_add_axiom :
    forall (f : Σ → 𝓡 Σ') I (R : I → 𝓡 Σ),
      lift f (add R) ⟺ add (fun i, lift f (R i)).
  Proof. (* 21 lines *) Qed.
```

```
    Theorem lift_single_axiom :
      forall (f : Σ → ℛ Σ') (s : Σ),
        lift f (single s) ⟺ f s.
    Proof. (* 19 lines *) Qed.

    Theorem lift_mul_axiom :
      forall (f : Σ → ℛ Σ') I (R : I → ℛ Σ),
        lift f (mul R) ⟺ mul (fun i, lift f (R i)).
    Proof. (* 15 lines *) Qed.

  End StateSection.

End WeakestPreconditionOperators.

Module WeakestPreconditionExtra.

  Import WeakestPreconditionDefinitions
         WeakestPreconditionAxioms
         WeakestPreconditionOperators.

  Module RAT :=
    RATheorems.Make
      (WeakestPreconditionDefinitions)
      (WeakestPreconditionAxioms).
  Import RAT.

  Module RAOP :=
    RAOperators.Make
      (WeakestPreconditionDefinitions)
      (WeakestPreconditionAxioms)
      (WeakestPreconditionOperators).

  Module RAOT :=
    RAOperatorTheorems.Make
      (WeakestPreconditionDefinitions)
      (WeakestPreconditionAxioms)
      (WeakestPreconditionOperators).

  Module RAN := RANotations.Make( WeakestPreconditionDefinitions ).
  Import RAN.

  Theorem lift_equiv_default : forall {Σ Σ'} (f : Σ → ℛ Σ') (R : ℛ Σ),
    lift f R ⟺
      ⊕ (fun S : { S | S ⊨ R } =>
        ⊗ (fun σ : From (proj1_sig S) =>
          f (proj1_sig σ))).
  Proof. (* 4 lines *) Qed.

  Theorem bind_equiv_default :
    forall {Σ Σ' Σ"} (f : Σ → ℛ Σ') (g : Σ' → ℛ Σ"),
      f_equiv (bind f g) (RAOP.primitive_bind f g).
  Proof. (* 9 lines *) Qed.

End WeakestPreconditionExtra.
```

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Amighi, A., Blom, S., Huisman, M., and Zaharieva-Stojanovski, M. The VerCors project: Setting up Basecamp. In *PLPV* (2012), K. Claessen and N. Swamy, Eds., ACM, pp. 71–82.

[3] Appel, A. W. Verified Software Toolchain - (Invited Talk). In *ESOP* (2011), G. Barthe, Ed., vol. 6602 of *Lecture Notes in Computer Science*, Springer, pp. 1–17.

[4] Appel, A. W. VeriSmall: Verified Smallfoot Shape Analysis. In *CPP* (2011), J.-P. Jouannaud and Z. Shao, Eds., vol. 7086 of *Lecture Notes in Computer Science*, Springer, pp. 231–246.

[5] Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *TPHOLs* (2005), J. Hurd and T. F. Melham, Eds., vol. 3603 of *Lecture Notes in Computer Science*, Springer, pp. 50–65.

[6] Ball, T., Hackett, B., Lahiri, S. K., Qadeer, S., and Vanegue, J. Towards Scalable Modular Checking of User-Defined Properties. In *VSTTE* (2010), G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, Eds., vol. 6217 of *Lecture Notes in Computer Science*, Springer, pp. 1–24.

[7] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A Modular Reusable Verifier for Object-oriented Programs. In *FMCO 2005, volume 4111 of LNCS* (2006), Springer, pp. 364–387.

[8] Barnett, M., and Leino, K. R. M. Weakest-precondition of unstructured programs. In *PASTE* (2005), M. D. Ernst and T. P. Jensen, Eds., ACM, pp. 82–87.

[9] Barnett, M., Leino, K. R. M., and Schulte, W. The Spec# programming system: An overview. In *Proceedings of CASSIS 04* (2004), Springer, pp. 49–69.

[10] Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., and Requet, A. JACK - A Tool for Validation of Security and Behaviour of Java Applications. In *FMCO* (2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4709 of *Lecture Notes in Computer Science*, Springer, pp. 152–174.

[11] Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. Shape Analysis for Composite Data Structures. In Damm and Hermanns [35], pp. 178–192.

[12] Berdine, J., Calcagno, C., and O'Hearn, P. W. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO* (2005), F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111 of *Lecture Notes in Computer Science*, Springer, pp. 115–137.

[13] Berdine, J., Calcagno, C., and O'Hearn, P. W. Symbolic Execution with Separation Logic. In *APLAS* (2005), K. Yi, Ed., vol. 3780 of *Lecture Notes in Computer Science*, Springer, pp. 52–68.

[14] Berdine, J., Cook, B., and Ishtiaq, S. SLAyer: Memory Safety for Systems-Level Code. In *CAV* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 178–183.

[15] Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., Eds. *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings* (2009), vol. 5674 of *Lecture Notes in Computer Science*, Springer.

[16] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[17] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages* (Wrocław, Poland, August 2011).

[18] Boogie: An intermediate verification language. http://research.microsoft.com/en-us/projects/boogie/.

[19] Borras, and Cari. Overexposure of radiation therapy patients in panama: problem recognition and follow-up measures. *Revista Panamericana de Salud Pública/Pan American Journal of Public Health 20*, 2-3 (Sept. 2006), 173–187.

[20] Botincan, M., Distefano, D., Dodds, M., Grigore, R., and Parkinson, M. J. coreStar: The Core of jStar. In *Boogie* (2011), pp. 65–77.

[21] Botincan, M., Dodds, M., Donaldson, A. F., and Parkinson, M. J. Automatic Safety Proofs for Asynchronous Memory Operations. In *PPOPP* (2011), C. Cascaval and P.-C. Yew, Eds., ACM, pp. 313–314.

[22] Botincan, M., Dodds, M., Donaldson, A. F., and Parkinson, M. J. Safe Asynchronous Multicore Memory Operations. In *ASE* (2011), P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds., IEEE, pp. 153–162.

[23] Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., and Rinard, M. Using First-Order Theorem Provers in the Jahob data structure verification system. In *Verification, Model Checking and Abstract Interpretation* (November 2007), vol. 4349 of *LNCS*.

[24] Box, D. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2002.

[25] Boyland, J. Checking Interference with Fractional Permissions. In *SAS* (2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer, pp. 55–72.

[26] Bozdag, E. Therac-25 and the security of the computer controlled equipment Ethics of Science and Technology.

[27] Calcagno, C., and Distefano, D. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods* (2011), M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617 of *Lecture Notes in Computer Science*, Springer, pp. 459–465.

[28] Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. Compositional Shape Analysis by Means of Bi-abduction. In *POPL* (2009), Z. Shao and B. C. Pierce, Eds., ACM, pp. 289–300.

[29] Chlipala, A. Certified programming with dependent types. http://adam.chlipala.net/cpdt/.

[30] Clarke, D. G., Potter, J., and Noble, J. Ownership Types for Flexible Alias Protection. In *OOPSLA* (1998), B. N. Freeman-Benson and C. Chambers, Eds., ACM, pp. 48–64.

[31] Cohen, E., Dahlweid, M., Hillebrand, M. A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A Practical System for Verifying Concurrent C. In Berghofer et al. [15], pp. 23–42.

[32] Cok, D. R., and Kiniry, J. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS* (2004), G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362 of *Lecture Notes in Computer Science*, Springer, pp. 108–128.

[33] The Coq Proof Assistant. http://coq.inria.fr/.

[34] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., and Schulte, W. VCC: Contract-based Modular Verification of Concurrent C. In *ICSE Companion* (2009), IEEE, pp. 429–430.

[35] Damm, W., and Hermanns, H., Eds. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings* (2007), vol. 4590 of *Lecture Notes in Computer Science*, Springer.

[36] David, C., and Chin, W.-N. Immutable Specifications for more Concise and Precise Verification. In *OOPSLA* (2011), C. V. Lopes and K. Fisher, Eds., ACM, pp. 359–374.

[37] de Moura, L. M., and Bjørner, N. Z3: An Efficient SMT Solver. In *TACAS* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.

[38] DeLine, R., and Leino, K. R. M. Boogiepl: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research, Mar. 2005.

[39] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: a Theorem Prover for Program Checking. *J. ACM 52*, 3 (2005), 365–473.

[40] Detlefs, D. L., Rustan, and Nelson, G. Wrestling with rep exposure. Tech. rep., 1998.

[41] Dietl, W. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Ph.D., Department of Computer Science, ETH Zurich, Dec. 2009. Doctoral Thesis ETH No. 18522.

[42] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, 1976.

[43] Distefano, D. Attacking Large Industrial Code with Bi-abductive Inference. In *FMICS* (2009), M. Alpuente, B. Cook, and C. Joubert, Eds., vol. 5825 of *Lecture Notes in Computer Science*, Springer, pp. 1–8.

[44] Distefano, D., O'Hearn, P. W., and Yang, H. A Local Shape Analysis Based on Separation Logic. In *TACAS* (2006), H. Hermanns and J. Palsberg, Eds., vol. 3920 of *Lecture Notes in Computer Science*, Springer, pp. 287–302.

[45] Distefano, D., and Parkinson, M. J. jStar: Towards Practical Verification for java. In *OOPSLA* (2008), G. E. Harris, Ed., ACM, pp. 213–226.

[46] Dudka, K., Müller, P., Peringer, P., and Vojnar, T. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures - (Competition Contribution). In *TACAS* (2012), C. Flanagan and B. König, Eds., vol. 7214 of *Lecture Notes in Computer Science*, Springer, pp. 545–548.

[47] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*, 6 ed. June 2012.

[48] Erlang programming language. http://www.erlang.org/.

[49] Filliâtre, J.-C. Verifying two lines of C with Why3: an exercise in program verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)* (Philadelphia, USA, January 2012).

[50] Filliâtre, J.-C., and Marché, C. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Damm and Hermanns [35], pp. 173–177.

[51] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended Static Checking for Java. In *PLDI* (2002), J. Knoop and L. J. Hendren, Eds., ACM, pp. 234–245.

[52] Flanagan, C., and Saxe, J. B. Avoiding exponential explosion: Generating compact verification conditions. In *POPL* (2001), C. Hankin and D. Schmidt, Eds., ACM, pp. 193–205.

[53] Frama-c. http://frama-c.com/index.html.

[54] Gotsman, A., Berdine, J., and Cook, B. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS* (2006), K. Yi, Ed., vol. 4134 of *Lecture Notes in Computer Science*, Springer, pp. 240–260.

[55] Gupta, A., and Malik, S., Eds. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings* (2008), vol. 5123 of *Lecture Notes in Computer Science*, Springer.

[56] Haack, C., Huisman, M., and Hurlin, C. Permission-Based Separation Logic for Multithreaded Java Programs. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica 15* (2011), 13–23.

[57] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM 12*, 10 (1969), 576–580.

[58] Hoare, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques* (1972), vol. 9 of *A.P.I.C. Studies in Data Processing*, Academic Press, pp. 61–71.

[59] Hol4. http://hol.sourceforge.net/.

[60] Homeier, P., Gunter, C. A., and Gunter, E. L. Sunrise homepage. http://www.cis.upenn.edu/~hol/sunrise/.

[61] Homeier, P. V., and Martin, D. F. Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator. In *TPHOLs* (1994), T. F. Melham and J. Camilleri, Eds., vol. 859 of *Lecture Notes in Computer Science*, Springer, pp. 269–284.

[62] Homeier, P. V., and Martin, D. F. A Mechanically Verified Verification Condition Generator. *Comput. J. 38*, 2 (1995), 131–141.

[63] Homeier, P. V., and Martin, D. F. Mechanical Verification of Mutually Recursive Procedures. In *CADE* (1996), M. A. McRobbie and J. K. Slaney, Eds., vol. 1104 of *Lecture Notes in Computer Science*, Springer, pp. 201–215.

[64] Homeier, P. V., and Martin, D. F. Mechanical Verification of Total Correctness through Diversion Verification Conditions. In *TPHOLs* (1998), J. Grundy and M. C. Newey, Eds., vol. 1479 of *Lecture Notes in Computer Science*, Springer, pp. 189–206.

[65] Howard, W. A. *The formulae-as-types notion of construction*. Academic Press, London-New York, 1980, pp. 480–490.

[66] Jacobs, B., and Piessens, F. Expressive Modular Fine-grained Concurrency Specification. In *POPL* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 271–282.

[67] Jacobs, B., Smans, J., and Piessens, F. A Quick Tour of the VeriFast Program Verifier. In *APLAS* (2010), K. Ueda, Ed., vol. 6461 of *Lecture Notes in Computer Science*, Springer, pp. 304–311.

[68] Jacobs, B., Smans, J., and Piessens, F. Verification of Unloadable Modules. In *FM* (2011), M. Butler and W. Schulte, Eds., vol. 6664 of *Lecture Notes in Computer Science*, Springer, pp. 402–416.

[69] Jones, C. B. Specification and Design of (Parallel) Programs. In *IFIP Congress* (1983), pp. 321–332.

[70] Kassios, I. T. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM* (2006), J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085 of *Lecture Notes in Computer Science*, Springer, pp. 268–283.

[71] Kassios, I. T. *A theory of object oriented refinement*. PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 2006.

[72] Kassios, I. T. The Dynamic Frames Theory. *Formal Aspects of Computing 23*, 3 (2011), 267–289.

[73] Kassios, I. T., Müller, P., and Schwerhoff, M. Comparing Verification Condition Generation with Symbolic Execution: An Experience Report. In *VSTTE* (2012), R. Joshi, P. Müller, and A. Podelski, Eds., vol. 7152 of *Lecture Notes in Computer Science*, Springer, pp. 196–208.

[74] Abstract on Korean Air Flight 801 Conclusions, Probable Cause, and Safety Recommendations. NTSB/AAR-99/02, August 1997.

[75] King, J. C. Symbolic Execution and Program Testing. *Commun. ACM 19*, 7 (1976), 385–394.

[76] Lahiri, S. K., and Qadeer, S. Verifying properties of well-founded linked lists. In *POPL* (2006), J. G. Morrisett and S. L. P. Jones, Eds., ACM, pp. 115–126.

[77] Leino, K. R. M. Efficient Weakest Preconditions. *Inf. Process. Lett. 93*, 6 (2005), 281–288.

[78] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (Dakar)* (2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer, pp. 348–370.

[79] Leino, K. R. M., and Müller, P. Object Invariants in Dynamic Contexts. In *ECOOP* (2004), M. Odersky, Ed., vol. 3086 of *Lecture Notes in Computer Science*, Springer, pp. 491–516.

[80] Leino, K. R. M., and Müller, P. A Basis for Verifying Multi-threaded Programs. In *ESOP* (2009), G. Castagna, Ed., vol. 5502 of *Lecture Notes in Computer Science*, Springer, pp. 378–393.

[81] Leino, K. R. M., Müller, P., and Smans, J. Verification of Concurrent Programs with Chalice. In *FOSAD* (2009), A. Aldini, G. Barthe, and R. Gorrieri, Eds., vol. 5705 of *Lecture Notes in Computer Science*, Springer, pp. 195–222.

[82] Leino, R., and Schulte, W. A Verifying Compiler for a Multi-threaded Object-oriented Language. Marktoberdorf lecture notes, 2007. In Manfred Broy, Johannes Grünbauer, Tony Hoare (eds.). Software System Reliability and Security. IOS Press, 2007.

[83] Leroy, X. Formal Verification of a Realistic Compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[84] Leveson, N. G. An Investigation of the Therac-25 Accidents. *IEEE Computer 26* (1993), 18–41.

[85] M., N. Crash of American Airlines Boeing, December 1995.

[86] Magill, S., Berdine, J., Clarke, E. M., and Cook, B. Arithmetic Strengthening for Shape Analysis. In *SAS* (2007), H. R. Nielson and G. Filé, Eds., vol. 4634 of *Lecture Notes in Computer Science*, Springer, pp. 419–436.

[87] Magill, S., Tsai, M.-H., Lee, P., and Tsay, Y.-K. THOR: A Tool for Reasoning about Shape and Arithmetic. In Gupta and Malik [55], pp. 428–432.

[88] McCarthy, J., and Hayes, P. J. Readings in nonmonotonic reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987, ch. Some philosophical problems from the standpoint of artificial intelligence, pp. 26–45.

[89] Moggi, E. Notions of Computation and Monads. *Inf. Comput. 93*, 1 (1991), 55–92.

[90] Moy, Y. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Jan. 2009.

[91] Müller, P. *Modular Specification and Verification of Object-oriented Programs*. Springer-Verlag, Berlin, Heidelberg, 2002.

[92] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local Reasoning about Programs that Alter Data Structures. In *CSL* (2001), L. Fribourg, Ed., vol. 2142 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[93] The Mozart Programming System. http://www.mozart-oz.org/.

[94] Parkinson, M. J., and Bierman, G. M. Separation Logic and Abstraction. In *POPL* (2005), J. Palsberg and M. Abadi, Eds., ACM, pp. 247–258.

[95] Philippaerts, P., Vogels, F., Smans, J., Jacobs, B., and Piessens, F. The Belgian Electronic Identity Card: a Verification Case Study. *ECEASST 46* (2011).

[96] Pierce, B. C., Casinghino, C., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. Software foundations. http://www.cis.upenn.edu/~bcpierce/sf/, July 2012.

[97] Poplmark. http://www.seas.upenn.edu/~plclub/poplmark/.

[98] Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS* (2002), IEEE Computer Society, pp. 55–74.

[99] Roy, P. V., and Haridi, S. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[100] Schulte, W., Xia, S., Smans, J., and Piessens, F. A Glimpse of a Verifying C Compiler Extended Abstract, 2007.

[101] Schwarz, J. Generic Commands - A Tool for Partial Correctness Formalisms. *Comput. J. 20*, 2 (1977), 151–155.

[102] Schwerhoff, M. Symbolic Execution for Chalice. Master's thesis, ETH Zurich, 2011.

[103] Smans, J., Jacobs, B., and Piessens, F. Symbolic Execution for Implicit Dynamic Frames. http://people.cs.kuleuven.be/~jan.smans/oopsla09.pdf.

[104] Smans, J., Jacobs, B., and Piessens, F. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP* (2009), S. Drossopoulou, Ed., vol. 5653 of *Lecture Notes in Computer Science*, Springer, pp. 148–172.

[105] Smans, J., Jacobs, B., and Piessens, F. Heap-Dependent Expressions in Separation Logic. In *FMOODS/FORTE* (2010), J. Hatcliff and E. Zucca, Eds., vol. 6117 of *Lecture Notes in Computer Science*, Springer, pp. 170–185.

[106] Smans, J., Jacobs, B., and Piessens, F. Implicit Dynamic Frames. *ACM Transactions on Programming Languages and Systems 34*, 1 (May 2012), 2:1–2:58.

[107] Strecker, M. Formal Verification of a Java Compiler in Isabelle. In *CADE* (2002), A. Voronkov, Ed., vol. 2392 of *Lecture Notes in Computer Science*, Springer, pp. 63–77.

[108] Tuerk, T. A Formalisation of Smallfoot in HOL. In Berghofer et al. [15], pp. 469–484.

[109] Villard, J., Lozes, É., and Calcagno, C. Tracking Heaps That Hop with Heap-Hop. In *TACAS* (2010), J. Esparza and R. Majumdar, Eds., vol. 6015 of *Lecture Notes in Computer Science*, Springer, pp. 275–279.

[110] Vogels, F. Companion Coq Scripts. http://people.cs.kuleuven.be/~frederic.vogels/coq-scripts.zip.

[111] Vogels, F., Jacobs, B., and Piessens, F. A machine checked soundness proof for an intermediate verification language: extended version. CW Reports CW526, Department of Computer Science, K.U.Leuven, Oct. 2008.

[112] Vogels, F., Jacobs, B., and Piessens, F. A Machine-checked Soundness Proof for an Efficient Verification Condition Generator. In *SAC* (2010), S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds., ACM, pp. 2517–2522.

[113] Wadler, P. The Essence of Functional Programming. In *POPL* (1992), R. Sethi, Ed., ACM Press, pp. 1–14.

[114] Wong, W., Debroy, V., and A., R. The role of software in recent catastrophic accidents. *IEEE Transactions on Reliability 59*, 3 (September 2010), 469–473.

[115] Wong, W. E., Debroy, V., Surampudi, A., Kim, H., and Siok, M. F. Recent catastrophic accidents: Investigating how software was responsible. In *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on* (June 2010), IEEE, pp. 14–22.

[116] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. W. Scalable Shape Analysis for Systems Code. In Gupta and Malik [55], pp. 385–398.

Appendix E

Publications

E.1 Papers at international conferences and symposia, published in full in proceedings

- Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 570–581. Springer, 2009.
- Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 2517–2522. ACM, 2010.
- Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. Annotation inference for separation logic based verifiers. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2011.
- Pieter Philippaerts, Frédéric Vogels, Jan Smans, Bart Jacobs, and Frank Piessens. The Belgian Electronic Identity Card: a Verification Case Study. *ECEASST*, 46, 2011.
- Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods,*, pages 41–55. Springer, 2011.

E.2 Internal reports

- Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language: extended version. CW Reports CW526, Department of Computer Science, K.U.Leuven, October 2008.
- Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator: technical report. CW Reports CW568, Department of Computer Science, K.U.Leuven, April 2010.
- Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight VeriFast: Extended version. CW Reports CW614, Department of Computer Science, KU Leuven, January 2012.