KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Advanced Semantics for Non-deterministic and Probabilistic Programming

Alexander Vandenbroucke

Supervisor:
Prof. dr. ir. Tom Schrijvers

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2020

Advanced Semantics for Non-deterministic and Probabilistic Programming

Alexander VANDENBROUCKE

Examination committee:
Prof. dr. ir. Hendrik Van Brussel, chair
Prof. dr. ir. Tom Schrijvers, supervisor
Prof. dr. ir. Frank Piessens
Prof. dr. ir. Gerda Janssens

Prof. dr.Maciej Piróg
(University of Wrocław)
Prof. dr.Alexandra Silva
(University College London)

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2020

© 2020 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Alexander Vandenbroucke, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

“ Timeo Danaos et dona ferentes.[1] *”*

— *Virgil, Aeneid II, 49*

This thesis roughly spans five and a half years of work. Over that time I have felt grateful to many people on numerous occasions. Therefore, I would like to take this opportunity to express my gratitude to several of them.

Foremost, I owe my thanks to my supervisor, Tom Schrijvers. It is his advice and support which have moulded me into the researcher that I am today. It has been an honour and a pleasure to work with such a brilliant mind.

I would like to thank the jury members, Frank Piessens, Gerda Janssens, Maciej Piróg and Alexandra Silva for reading and revising the text, and for their insightful comments and questions during my defence.

These past five years, I have been most fortunate to work in a diverse and pleasant professional environment. Thank you for being my office mates and colleagues, Steven Keuchel, Amr Saleh, Klara Marntirosian, Gert-Jan Bottu, Ruben Pieters, Maciej Piróg, César Santos, Koen Pauwels, Birthe van den Berg, Roger Bosman, Matthias van der Hallen, and finally to George Karachalias, who diligently prepared us all for amazement, and whose practical advise was indispensable during the writing of this thesis.

I am grateful to my friends, who never hesitate to remind me that my research is funded by the government, that is, the tax payer, and thus have also made a material contribution to this work.

Finally, I am thankful to my parents and my sister for their support, and also to my uncle, Tim Heysse, who, when I was discussing whether or not to pursue a PhD, replied “Why not?”—a philosophical answer, if ever there was one.

[1] I fear the Greeks, especially when they bear gifts.

Abstract

Declarative programming is the idea that a program should describe *what* it does, *which* problem it is solving, not *how* it does so. In this fashion, declarative programming focuses on the complexity that is inherent in the problem, not the accidental complexity that originates from solving the problem on a computer. By reducing accidental complexity, declarative programming eliminates a common source of bugs. This thesis contributes towards declarative programming in three distinct, but partly overlapping paradigms: *logic programming*, *functional programming* and *probabilistic programming*.

The first part of this thesis studies tabling, which is a powerful resolution mechanism for logic programming that ensures termination of a larger class of programs. Extensions of this technique, such as mode-directed tabling and answer subsumption allow efficient tabling for optimisation problems. While much attention has been devoted to the expressivity and efficiency of these approaches, soundness has not been considered. This thesis shows that the different implementations indeed fail to produce the correct answer for some programs. To remedy this situation, the thesis provides a formal framework that generalises the existing approaches and establishes a correctness criterion that explains when the approach is sound.

Then, tabling is adapted for non-determinism in the functional programming language Haskell. Non-determinism is a technique that allows expressing a wide range of problems in a concise and declarative manner. Essentially, non-determinism means that programs can return zero or more answers as opposed to a single deterministic one. However, Haskell's default recursion mechanism is not the right one for non-deterministic programs, and causes non-termination. Instead, the correct semantics is a set-based least fixed point semantics, which is implemented via tabling as a monad. The set-based least fixed point semantics is then generalised to arbitrary complete lattices, enabling tabling for optimisation problems.

The second part of this thesis presents P$\lambda\omega$NK, a probabilistic programming language for modelling networks. Unlike the earlier modelling language Probabilistic NetKAT (PNK), P$\lambda\omega$NK provides higher-order functions, to make programming much more convenient and declarative.

The formalisation of P$\lambda\omega$NK is challenging for two reasons: Firstly, network programming introduces multiple side effects (in particular, non-determinism and probabilistic choice) which need to be carefully controlled in a functional setting. For this purpose, the system uses an explicit syntax for functions and sequencing which makes the interplay of these effects explicit. Secondly, measure theory, the standard domain for formalisation of probabilistic languages does not admit higher-order functions. We address this by leveraging ω-Quasi Borel Spaces, a recent advancement in the domain theory of probabilistic programming.

This work is not only useful for bringing abstraction to PNK. The meta-theory for P$\lambda\omega$NK, which combines advanced features like higher-order functions, iteration and parallelism, may also inform similar meta-theoretic efforts.

Nederlandstalig Abstract

Declaratief programmeren omvat de idee dat een programma hoort te beschrijven *wat* het doet, niet *hoe* het dit doet. Op deze manier focust declaratief programmeren zich op de complexiteit die inherent is aan het probleem, niet de bijkomstige complexiteit die zijn oorsprong vindt in het oplossen van het probleem op een computer. Dankzij het verminderen van de bijkomstige complexiteit kan het declaratief programmeren een vaak voorkomende bron van bugs elimineren. Deze thesis levert bijdragen aan declaratief programmeren in drie verschillende, maar deels overlappende paradigma's: *logisch programmeren*, *functioneel programmeren* en *probabilistisch programmeren*.

Het eerste deel van deze thesis bestudeert tabulatie (tabling), een krachtig resolutiemechanisme voor logisch programmeren, dat een eindige looptijd verzekert voor een grotere klasse programma's dan de traditionele procedures. Uitbreidingen van deze techniek, zoals modus-gestuurde tabulatie (mode-directed tabling) en antwoordsubsumptie (answer subsumption) laten bovendien efficiënte tabulatie voor optimalisatieproblemen toe. In tegenstelling de expressiviteit en efficiëntie van deze methodes is de correctheid ervan echter nauwelijks onderzocht. Deze thesis toont aan dat bestaande implementaties in sommige gevallen inderdaad niet het juiste resultaat produceren. Om deze situatie het hoofd te bieden, bouwt deze thesis een formeel raamwerk, dat de bestaande uitbreidingen veralgemeent. Binnen dit raamwerk geldt een correctheidsconditie die uitlegt wanneer het juiste resultaat berekend wordt.

Daarnaast wordt tabulatie aangepast voor niet-determinisme in de functionele programmeertaal Haskell. Niet-determinisme is een techniek waarmee een breed scala aan problemen op een summiere en declaratieve manier kunnen worden uitgedrukt. In essentie betekent niet-determinisme dat een programma nul of meerdere antwoorden kan teruggeven, in plaats van één enkel deterministisch resultaat. Echter, het standaard recursiemechanisme in Haskell is niet geschikt voor niet-deterministische programma's, en kan het uitvoeren van een programma binnen een eindige looptijd verhinderen. De

juiste semantiek wordt gegeven door het kleinste vaste punt (least fixed point) in de verzamelingentralie (powerset lattice) te bereken. Dit wordt door middel van tabulatie geïmplementeerd als een monad. Deze semantiek op basis van verzamelingen wordt veralgemeend naar arbitraire volledige tralies (complete lattices), wat tabulatie voor optimalisatieproblemen in Haskell mogelijk maakt.

Het tweede deel van deze thesis beschrijft P$\lambda\omega$NK, een probabilistische programmeertaal voor het modelleren van netwerken. In tegenstelling tot de eerdere taal Probabilistic NetKAT (PNK) bevat P$\lambda\omega$NK hogere-orde functies, zodat programmeren aangenamer en declaratiever is.

De formalisatie van P$\lambda\omega$NK is uitdagend omwille van twee redenen: Ten eerste introduceert het modelleren van netwerken meerdere neveneffecten (in het bijzonder, niet-determinisme en probabilistische keuze) die in een functionele context zorgvuldig in bedwang moeten worden gehouden. Het P$\lambda\omega$NK-systeem maakt hiertoe gebruik van een expliciete syntaxis voor functies en opeenvolgende acties die de interactie van de effecten expliciet maakt. Ten tweede, de maattheorie (measure theory), het standaard domein voor de formalisatie van probabilistische talen, is niet geschikt voor hogere-orde functies. Dit probleem wordt opgelost door gebruik te maken van ω-Quasi Borel Ruimten (ω-Quasi Borel Spaces), een recente ontwikkeling in de domeintheorie voor probabilistisch programmeren.

Tenslotte is dit werk niet enkel nuttig omdat het abstractie toevoegt aan PNK; maar ook omdat de metatheorie voor P$\lambda\omega$NK geavanceerde concepten zoals hogere-orde functies, iteratie en parallellisme combineert, en dus ook bij gelijkaardige metatheoretische ontwikkelingen behulpzaam kan zijn.

Contents

Abstract iii

Nederlandstalig Abstract v

Contents vii

1 Introduction 1
1.1 Declarative Programming 2
1.2 Thesis Overview 3

I Tabling 7

2 An Introduction to Tabling 9
2.1 The Semantics of Tabling 11

3 Sound Answer Subsumption 15
3.1 Answer Subsumption 15
3.2 The Semantics of Answer Subsumption 16
3.3 Generalised Answer Subsumption Semantics 25
3.4 Historical Notes and Related Work 28
3.5 Proof of Monotonicity of T_P^L 31
3.6 Scientific output 31

4 Monadic Tabling in Haskell 33
4.1 Non-determinism 35
4.2 Effect Handlers 37
4.3 Explicating Recursion 39
4.4 Effect Handler for Explicit Recursion 40
4.5 Dependency Tracking 45
4.6 Lattices 48

4.7 Mutual Recursion 61
4.8 Benchmarks 66
4.9 Related Work 67
4.10 Scientific Output 70

II Probabilistic Programming for Modelling Networks 71

5 Probabilistic Network Modelling 73

6 Functional Probabilistic NetKAT 79
6.1 A Brief Introduction to Probabilistic NetKAT 79
6.2 Modelling in PNK 80
6.3 Extending PNK with functions 82
6.4 Syntax of Terms 85
6.5 Types and Type System 87
6.6 Related Work 89

7 A Convenient Category for P$\lambda\omega$NK 91
7.1 ω-Complete Partial Orders 92
7.2 Quasi-Borel Spaces 93
7.3 ω-Quasi-Borel Spaces 94
7.4 A Commutative Probabilistic Powerdomain 95

8 Denotational Semantics for P$\lambda\omega$NK 97
8.1 Semantics of Types 98
8.2 Semantics of Terms 101
8.3 Finite Maps 102
8.4 Continuity 104
8.5 Conservativity 105
8.6 Related Work 106

9 Compilation to PNK 107
9.1 Elaboration 108
9.2 Reduction 109
9.3 Meta-theory of Reduction 110
9.4 Discussion 112
9.5 Example 112

10 Conclusion 117
10.1 Goal of the Thesis 117
10.2 Summary of Part I 118

10.3 Summary of Part II 119
10.4 Future Work 120
10.5 Concluding Remarks 121

A Additional material for P$\lambda\omega$NK **125**
A.1 Atomic Computations 125
A.2 Re-translation of PNK semantics 127
A.3 Proofs 128

Index **165**

Bibliography **167**

List of Publications **175**

Chapter 1

Introduction

> *“ ‘So Hex here has caught daftness off the Bursar,’ said Ridcully. ‘Simple. Real stupidity beats artificial intelligence every time.’ ”*
>
> — *Terry Pratchett, Hogfather*

The best code is the code that never gets written. This idea drives programmers and programming language researchers to develop techniques and languages that reduce their programs' *accidental* complexity. That is, complexity that is not essential to the problem at hand, but rather originates from having to solve that problem on a fairly unintelligent electronic calculator. For example, a programmer may need to manually allocate and free memory, carefully update explicit indices inside a loop to iterate over collections, or manipulate complex intermediate data structures at every turn.

These complications will sound familiar to anyone who has had the character-building experience of writing a typical imperative program. They are a frequent source of bugs, either directly or indirectly, because brain cycles spent on ancillary issues cannot be devoted to the real problem instead. Furthermore, from critical infrastructure, manufacturing, and finance, to online shopping and “smart” refrigerators; our society is becoming ever more dependent on ever more complex software. Thus, we must expend every effort to make our software more reliable.

In contrast to imperative programming, a declarative program avoids accidental complexity by not caring how a problem is solved. It simply states the problem's parameters and the objective it is trying to solve, declaring: “*This* is the problem I want you to solve; *how* you go about it, I don't care.” Of course, the complete

elimination of accidental complexity is typically unachievable in practice, but some programming languages and paradigms come closer than others.

1.1 Declarative Programming

A minimal level of declarative programming is found in most modern programming languages: for instance, automatic memory management and structured programming (no GOTOs) have become the norm—except when extreme performance is desired. Other declarative conveniences have long since become mainstream, e.g., iterable collections. Moreover, certain paradigms are particularly well-suited to declarative programming: functional programming, logic programming and probabilistic programming—although even here operational impediments can rear their ugly heads.

Functional programming languages (e.g., Haskell, OCaml, and LISP with its many dialects) treat functions as first-class values. This decreases the impedance mismatch between a mathematical model of the problem and the program that implements that model. Lazy functional programming further narrows the gap by deviating from the usual strict evaluation order of most programming languages, to allow more compositional programs, and more powerful equational reasoning.

Logic programs use predicate logic to describe the properties of the problem domain. The logic programming system (e.g., a Prolog engine) then uses search algorithms to find a solution that is consistent with the logical theory. Because logic lacks the vocabulary to talk about the modalities of the search, logic programming excels at separating the properties of the problem domain (the logic) from the search (the control). It obviates the need to write the search algorithm every time.

Probabilistic programming simplifies the process of probabilistic modelling. A probabilistic model usually consists of several interlocking probability distributions. The process of computing the probabilities of a model based on (real-world) data is called inference, a task performed by *inference algorithms*. In a traditional probabilistic model, the model and the inference algorithm, are closely intertwined. This makes it difficult to modify or reuse either the model or the algorithm separately. In contrast, a probabilistic program contains only the probabilistic model, written in a probabilistic programming language. Inference is performed by generic tunable inference algorithms that are provided with the probabilistic programming system. Examples (by no means exhaustive) of such languages are Church, Anglican, ProbLog and probabilistic NetKAT.

1.2 Thesis Overview

This thesis makes contributions to two particular fields: tabling and probabilistic modelling of networks. In both these areas the goal is to narrow the gap between the declarative mathematical ideal and the messy reality of programming on a computer.

1.2.1 Part I: Tabling

Declarative programs often benefit from non-determinism: the idea that a problem can have zero or more solutions. This can lead to problems when writing recursive programs: naive recursion may fail to terminate even when the set of solutions is finite.

At the heart of the issue is an impedance mismatch between the formal least fixed point semantics of the program, and the recursion that is used to implement it. Tabling is a powerful mechanism that captures the least fixed point semantics of these programs more faithfully, thus allowing more programs to terminate. It derives its name from the central data structure, the *table*, wherein all solutions are stored. This allows tabling to detect when further recursion does not add new solutions, at which point execution can safely halt.

The first half of Part I studies the semantics of an advanced form of tabling called answer subsumption, also known as mode-directed tabling. While much attention has been paid to expressivity and efficiency of these approaches, soundness has not been considered. This thesis shows that the different implementations indeed fail to produce least fixed points for some programs. As a remedy, I provide a formal framework that generalises existing approaches, and I establish a soundness criterion that explains for which programs the approach is sound.

In the second half, I adapt the tabling approach for non-determinism in the purely functional programming language Haskell. Conventionally, non-determinism is implemented as lists of outcomes. This approach provides a concise and declarative description of certain problems. However, the traditional approach falls short when the problem is allowed to be recursive: the generated list of outcomes may be infinite. Yet there are usually only finitely many distinct *relevant* outcomes. This thesis shows that the set of relevant outcomes corresponds to a least fixed point, which may be computed in a finite amount of time by using tabling. Thus, non-determinism and recursion are allowed to co-exist meaningfully in a single program.

The first half (Chapters 2 and 3) draws material from the following publication:

Alexander Vandenbroucke, Maciej Piróg, Benoit Desouter, and Tom Schrijvers. 2016. Tabling with Sound Answer Subsumption. In Theory and Practice of Logic Programming. Cambridge University Press, 2016, Vol. 16, iss. 5, pp. 933–949. DOI: http://doi.org/10.1017/S147106841600048X

The second half (Chapter 4) draws material from the following publication:

Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. 2015. Fixing non-determinism. In Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (IFL '15). Association for Computing Machinery, New York, NY, USA, Article 5, 1-–12. DOI: https://doi.org/10.1145/2897336.2897342

1.2.2 Part II: Probabilistic Programming for Modelling Networks

Part II of the thesis presents P$\lambda\omega$NK, a functional probabilistic network programming language that extends Probabilistic NetKAT (PNK). Like PNK, it enables probabilistic modelling of network behaviour, by providing probabilistic choice and infinite iteration (to simulate looping network packets). Yet, unlike PNK, it also offers abstraction and higher-order functions to make programming much more convenient.

The formalisation of P$\lambda\omega$NK is challenging for two reasons: Firstly, network programming induces multiple side effects (in particular, parallelism and probabilistic choice) which need to be carefully controlled in a functional setting. The system uses a Fine-Grained Call-By-Value (FG-CBV) syntax which makes the interplay of these effects explicit. Secondly, measure theory, the standard domain for formalisations of (continuous) probabilistic languages, does not admit higher-order functions. I address this by leveraging ω-Quasi Borel Spaces (ωQBSes), a recent advancement in the domain theory of probabilistic programming languages.

I believe that this work is not only useful for bringing abstraction to PNK, but that—as part of our contribution—we have developed the meta-theory for a probabilistic language that combines advanced features like higher-order functions, iteration and parallelism, which may inform similar meta-theoretic efforts.

This part (Chapters 5, 6, 7, 8 and 9) draws material from the following publication:

Alexander Vandenbroucke and Tom Schrijvers. 2019. P$\lambda\omega$NK: functional probabilistic NetKAT. Proc. ACM Program. Lang. 4, POPL, Article 39 (January 2020), 27 pages. DOI: https://doi.org/10.1145/3371107

Part I

Tabling

Chapter 2

An Introduction to Tabling

This chapter reviews the motivation for tabling and the relevant theoretical background.

Tabling is a mechanism that makes more programs terminate. The advantages of tabling are perhaps best illustrated by means of an example:

Does the following Haskell program terminate?

```
p :: [(Int,Int)]
p = [(1,2)] ++ map swap p where swap (x,y) = (y,x)
```

It does not:

```
> p
[(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),...
```

Does the following program—a Prolog rewrite of the previous one—terminate?

```
p(1,2).
p(X,Y) :- p(Y,X).
```

It does not terminate either:

```
?- p(X,Y).
X = 1, Y = 2;
X = 2, Y = 1;
X = 1, Y = 2;
...
```

Finally, does the same Prolog program terminate using *tabling*?

```
:- table p/2.
p(1,2).
p(X,Y) :- p(Y,X).
```

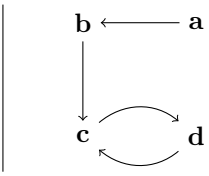
Third time is the charm:

```
?- p(X,Y).
X = 1, Y = 2;
X = 2, Y = 1.
```

The `table` directive in the third example tells the Prolog engine to use *tabled* execution on the predicate `p/2`. In tabled execution mode, the Prolog engine stores all the substitutions it has seen for `p/2` so far in a data structure, called a table. For example, in the program above, while trying to satisfy `p(X,Y)`, it first sees the fact `p(1,2).` and stores the substitution $X = 1, Y = 2$ in the table. Then, the Prolog engine backtracks and sees the rule `p(X,Y) :- p(Y,X).` Since `p/2` is tabled, it tries to resolve the recursive call `p(Y,X)` by looking in the table, producing the substitution $X = 2, Y = 1$, which is stored in the table as well. To complete the execution, the Prolog engine tries the recursive call again, with the new solution, this time producing the substitution $X = 1, Y = 2$, which is already in the table. As there are no new solutions, the call to `p(X,Y)` is complete. Note that the tabled program would still work if the order of the clauses is swapped, that is, tabling solves the usual issues of left-recursion and clause ordering as well.

Example 1. The usefulness of tabling becomes evident when applied to a more complex program:

```
edge(a,b). edge(b,c). edge(c,d). edge(d,c).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z,Y).
```



The `edge/2` predicate describes a directed cyclic graph, and `reachable/2` is the reachability predicate on that graph such that `reachable(X,Y)` succeeds if there is a path from `X` to `Y`. However, consider what happens on the query `?-reachable(a,b)`:

```
?- reachable(a,b).
true;
<stack-overflow>
```

The first `true` comes from the direct edge from `a` to `b`. However, upon backtracking, the predicate walks from `a` to `b`, then to `c`, to `d` and back to `c`. At this point, the predicate is stuck in a loop and will never visit `b` (in fact there's no path from `c` to `b`). If `reachable/2` is *tabled*, the Prolog Engine recognises that the loop does not reach any new nodes, and stops.

Tabling considerably improves the declarativity and expressiveness of the Prolog language. It eliminates the sensitivity of the traditional SLD resolution (Kowalski, 1974) to rule and goal ordering, allowing a larger class of programs to terminate. As an added bonus, the memoisation of the tabling mechanism may significantly improve run time performance in exchange for increased memory usage. Tabling has been implemented in several well-known Prolog systems, such as XSB (Swift and Warren, 2010, 2012),Yap (Santos Costa et al., 2012), Ciao (Chico de Guzmán et al., 2008), B-Prolog (Zhou, 2012), and SWI-Prolog (Wielemaker et al., 2012; Desouter et al., 2015) and has been successfully applied in various domains.

The remainder of this chapter establishes some of the basic notions and theory that underlie tabling in Prolog. Chapter 3 extends this theory to a more advanced form of tabling called *Answer Subsumption*. Chapter 4 returns to the very first example to study tabling in the context of Haskell. It then presents a monadic implementation of tabling.

2.1 The Semantics of Tabling

Because the operational semantics of tabling is rather complex and different systems vary in subtle ways, this thesis makes a simplifying assumption that tabling systems implement Lloyd's least fixed point semantics (Lloyd, 1984) for definite logic programs, that is, that tabling is a sound program optimisation with no impact on the denotation of a program. This semantics conveniently abstracts from low-level aspects such as clause and goal ordering and the specific clause scheduling algorithm used by the Prolog engine.

2.1.1 Least Fixed Point Semantics

First, we need the notion of a *Herbrand base*: the set of all possible (ground) atoms that occur in a logic program. More formally, let Σ be an alphabet, and P be a logic program over Σ, then the Herbrand base H_P is the set of all ground

atoms over Σ. For example, the Herbrand base of the reachability program is:

$$H_P = \{\texttt{edge(X,Y)} \mid X, Y \in \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}\}$$
$$\cup \{\texttt{reachable(X,Y)} \mid X, Y \in \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}\}$$

A *(Herbrand) interpretation* is a set $I \subseteq H_P$. Intuitively, it contains atoms in the Herbrand base that are true: $\forall a \in H_P : I \models a \iff a \in I$.

Finally, define the operator $T_P : \mathcal{P}(H_P) \to \mathcal{P}(H_P)$ such that, given an interpretation I, the value $T_P(I)$ is the interpretation that immediately follows from I by any of the program rules:

Definition 2 (Immediate Consequence operator)**.**

$$T_P(I) = \{B_0 \in H_P \mid B_0 \texttt{:-} B_1, \ldots, B_n \in \mathsf{ground}(P) \wedge \{B_1, \ldots, B_n\} \subseteq I\} \quad (2.1)$$

where $\mathsf{ground}(P)$ is the grounding of a program P.

This operator is called the *immediate consequence operator*. Its least fixed point with respect to subset-inclusion ($\subseteq$), denoted $\mathit{lfp}(T_P)$, defines the semantics of the program P, and is also known as the *least Herbrand model*. It is the interpretation that contains those and only those atoms that follow from the program.

Example 3. Consider the following program P:

```
p(a). p(b). q(c).
q(X) :- p(X).
```

Its Herbrand base is $\{\texttt{p(a)}, \texttt{p(b)}, \texttt{p(c)}, \texttt{q(a)}, \texttt{q(b)}, \texttt{q(c)}\}$. Its fixed point semantics is:

$$\mathit{lfp}(T_P) = \{\texttt{p(a)}, \texttt{p(b)}, \texttt{q(a)}, \texttt{q(b)}, \texttt{q(c)}\}$$

Observe that this is exactly the set of atoms that follow from the program.

2.1.2 Existence and Computability of the Least Herbrand Model

The least fixed point semantics is not necessarily well-defined: it is not immediate that the least fixed point actually exists. Moreover, if it exists, it may not actually be constructively computable.

Fortunately, there is no reason for concern: by appeal to a well-known theorem from lattice theory, we can easily establish the well-definedness. A *complete*

lattice is a partially ordered set (poset) $\langle L, \leq_L \rangle$ such that every $X \subseteq L$ has a least upper bound $\bigvee X$, i.e.:

$$\forall z \in L : \bigvee X \leq_L z \iff \forall x \in X : x \leq_L z$$

It follows from the definition that $\bigvee X$ is unique. A complete lattice always has a least element $\bot$ that is smaller than all other elements. It corresponds to the least upper bound of the empty set: $\bot = \bigvee \emptyset$. With a slight abuse of notation we also write $a \bigvee b$ to denote the least upper bound $\bigvee\{a, b\}$. We do indeed have a lattice structure at hand: the power set of the Herbrand base $\langle \mathcal{P}(H_P), \subseteq \rangle$ is a complete lattice. In fact, any power set is a complete lattice. Moreover, it is quite easy to see that if P is a definite logic program (i.e., contains no negations), then T_P is monotone with respect to this lattice. By the following theorem, it follows that $\mathit{lfp}(T_P)$ exists, ensuring that the semantics is well-defined for every definite program P:

Theorem 4 (Knaster–Tarski)**.** *Let $\langle L, \leq_L \rangle$ be a complete lattice, and let $f : L \to L$ be a monotone function. Then, the fixed points of f form a complete lattice. In particular, f has a least fixed point, denoted $\mathit{lfp}(f)$.*

Moreover, the T_P operator is ω-continuous, which means that for all ascending chains $l_1 \subseteq l_2 \subseteq \ldots$ with $l_1, l_2, \ldots \subseteq H_P$, it is the case that $\bigcup_{i=1}^{\infty} T_P(l_i) = T_P(\bigcup_{i=0}^{\infty} l_i)$. Then, Kleene's fixed point theorem[1] gives us a constructive way of obtaining $\mathit{lfp}(T_P)$:

$$\mathit{lfp}(T_P) = \bigcup\{T_P(\emptyset), T_P^2(\emptyset), \ldots\}$$

The least fixed point can therefore be obtained in a bottom-up fashion by iterating T_P from the empty set onward. Operationally, tabling usually interleaves a top-down goal-directed strategy with bottom-up iteration. The bottom-up strategy always computes the entire least Herbrand model, even when only a small portion of it may be required to prove a particular query. The top-down part of tabling avoids computing irrelevant atoms as much as possible, making inference feasible.

2.1.3 Stratification

Unfortunately, the T_P-operator is not monotone for programs containing more advanced constructs, such as negation. Therefore, Lloyd's semantics as described

[1] This theorem says that the least fixed point of an ω-continuous function f is precisely the least upper bound of the set $\{\bot, f(\bot), f(f(\bot), \ldots, f^n(\bot), \ldots\}$.

above is not suitable for capturing the semantics of such programs. In the case of negation, this problem is solved by partitioning the clauses of a program into an ordered set of *strata* based on their interdependence. This procedure is called *stratification* (Apt et al., 1988). Then, the semantics for each stratum is computed based on the semantics of the lower strata, with no relation to the higher strata. To make this more concrete, suppose a ground program P admits a stratification $P_1, \ldots, P_n$, with the P_i non-empty and pairwise disjoint, then:

$$P = P_1 \cup \cdots \cup P_n,$$

$$Q_1 = P_1,\ Q_{i+1} = P_{i+1} \cup M_i, \qquad \text{for all } i = 0, \ldots, n-1$$

$$M_i = \mathit{lfp}(T_{Q_i}) \qquad \text{for all } i = 1, \ldots n$$

where M_i should be understood as a set of facts. If a program admits a stratification where all negated calls are to predicates defined in lower strata, the obvious extension T_P^{neg} of the T_P operator to include negation is guaranteed to be monotone. The semantics of P is then given by $\bigcup_{i=1}^{n} M_i$.

Example 5. Consider the following variant of Example 1, appropriately divided in strata:

```
% Stratum P1:
edge(a,b). edge(b,c). edge(c,d). edge(d,c).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z),reachable(Z,Y).
% Stratum P2:
unreachable(X,Y) :- not(reachable(X,Y)).
% Stratum P3:
connected :- not(unreachable(X,Y)).
```

Least fixed point computation yields the following models (for brevity names of predicates have been shortened to their initial, e.g., `edge` is written `e`.):

$$\begin{aligned} M_1 &= \{\texttt{e(a,b)}, \texttt{e(b,c)}, \texttt{e(c,d)}, \texttt{e(d,c)}\} \\ &\cup \{\texttt{r(a,b)}, \texttt{r(a,c)}, \texttt{r(a,d)}, \texttt{r(b,c)}, \texttt{r(b,d)}, \texttt{r(c,d)}, \texttt{r(d,c)}\} \\ M_2 &= \{\texttt{u(c,a)}, \texttt{u(c,b)}, \texttt{u(d,a)}, \texttt{u(d,b)}\} \cup M_1 \\ M_3 &= \emptyset \cup M_2 = M_2 \end{aligned}$$

The final model does not contain `connected`, as expected.

Example 6. Consider the program:

```
even(0).
even(s(X)) :- odd(X).
odd(X) :- not(even(X)).
```

This program does not admit a stratification, because there is a cyclic dependency from `even` to itself (via `odd`) which passes through a negation.

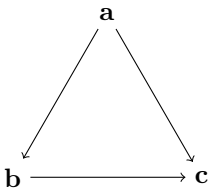
Chapter 3

Sound Answer Subsumption

3.1 Answer Subsumption

Much research effort has been devoted to improving the performance of tabling for various specialised use-cases (Swift, 1999; Ramakrishna et al., 1997; Zhou and Dovier, 2011). This chapter is concerned with one such fairly broad class of use-cases: we are not directly interested in all the answers to a tabled-predicate query, but instead wish to aggregate these answers somehow. The following shortest-path example illustrates this use-case.

```
query(X,Y,MinDist) :-
  findall(Dist,p(X,Y,Dist),List),
  min_list(List,MinDist).
:- table p/3.
p(X,Y,1) :- edge(X,Y).
p(X,Y,D) :-
  p(X,Z,D1), p(Z,Y,D2), D is D1 + D2.
edge(a,b). edge(b,c). edge(a,c).
```



The query `?- query(a,c,D).` computes the distance `D` of the shortest path from `a` to `c` by first computing the set of distances $\{1, 2\}$ of all paths and then selecting the smallest value from this set. Unfortunately, when the graph is cyclic, the set of distances is infinite and the query never returns, even though the infinite set has a well-defined minimal value.

Various tabling extensions, such as mode-directed tabling (Guo and Gupta, 2004, 2008; Zhou et al., 2010; Santos and Rocha, 2013), partial order answer subsumption and lattice answer subsumption (Swift and Warren, 2012), have

come up with ways to integrate the aggregation into the tabled resolution. This way answers are incrementally aggregated and the tabling may converge more quickly to the desired results. For instance, the shortest-path program can be written with mode-directed tabling as:

```
:- table p(+,+,min).
p(X,Y,1) :- edge(X,Y).
p(X,Y,D) :- p(X,Z,D1), p(Z,Y,D2), D is D1 + D2.
e(a,b). e(b,c). e(a,c).
```

Here the query `?- p(a,c,D).` yields only the shortest distance. It does so by greedily throwing away non-optimal intermediate results and in this way only considers finitely many paths, even if the graph is cyclic. In summary, this approach makes tabling (sometimes infinitely) more efficient for the aggregation use-case.

Unfortunately, none of the existing implementations that I am aware of is generally sound. Consider the following pure logic program.

```
p(0). p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

The query `?- p(X).` has a finite set of solutions, {`p(0)`,`p(1)`,`p(2)`,`p(3)`}, the largest of which is `p(3)`. However, XSB, Yap and B-Prolog all yield different (invalid) solutions when answer subsumption is used to obtain the maximal value. Both XSB and B-Prolog yield `X = 2`, with a maximum lattice and `max` table mode respectively. Yap (also with `max` table mode) yields `X = 0; X = 1; X = 2`, every solution except the right one.[1]

Clearly, these results are unsound. This example is not the only erroneous one; we can easily construct more erroneous scenarios with other supported forms of aggregation. Hence, we must conclude that answer subsumption is in general not a semantics-preserving optimisation. Yet, as far as I know, the existing literature does not offer any guidance on when the feature can be relied upon. In fact, to my knowledge, its semantics has not been formally discussed before.

3.2 The Semantics of Answer Subsumption

This section proposes a denotational semantics for tabling with answer subsumption. Broadly speaking, I modify the semantics for stratified programs

[1] The batch scheduling used by Yap returns any answer as soon as it is found.

as described in Chapter 2 in two respects. Firstly, I include new answers that may emerge from the program-defined rules of subsumption, which are not necessarily logical consequences of the same program without answer subsumption. I obtain this by extending the T_P operator. Secondly, it performs the actual subsumption, that is, it removes the subsumed answers. Stratification, as discussed in Section 3.2.4, is used to control the order in which these two steps are invoked.

In a bit more detail, the semantics of a stratum is given by the extended immediate consequence operator, written $\widehat{T}_P$, and a function $\eta : H_P \to L$ that aggregates the answers using a lattice L. A consequence of this specification is that an aggregation naturally ignores all operational aspects of the program P. That is to say, given two structurally distinct programs P_1 and P_2 whose least fixed point semantics coincide, that is, $lfp(\widehat{T}_{P_1}) = lfp(\widehat{T}_{P_2})$, it follows that $\bigvee \eta(lfp(\widehat{T}_{P_1})) = \bigvee \eta(lfp(\widehat{T}_{P_2}))$, i.e. their aggregates coincide as well.

Obviously, the existing systems do not implement answer subsumption as a single post-processing function. Instead, they execute it repeatedly during the bottom-up phase of the computation, which sometimes makes them deviate from the intended semantics. This behaviour is formalised and dealt with in Section 3.3.

For now, assume that the program P has only one stratum. Towards the end of the section, I show how to assemble the semantics of programs with any number of strata.

3.2.1 Mode-Directed Tabling

Mode-directed tabling is a convenient aggregation approach supported by ALS-Prolog, B-Prolog and Yap where the arguments of a tabled predicate are annotated with one of a range of aggregation *modes*. Yap provides the largest range of possible modes: `index`, `first`, `last`, `min`, `max` and `all`. The answers are grouped by distinct values for the `index` arguments; the remaining arguments are aggregated according to their mode—the mode names should be self-explanatory.

Based on our assumptions so far, existing implementations of the three modes `first`, `last` and `sum` are immediately disqualified. The reason is that the semantics of the existing implementations is inherently sensitive to the program structure. Consider the two programs below:

```
% P1
:- table p(first).
p(1). p(2).
```

```
% P2
:- table p(first).
p(2). p(1).
```

Clearly $lfp(\widehat{T}_{P1}) = \{\texttt{p(1)}, \texttt{p(2)}\} = lfp(\widehat{T}_{P2})$, however $P1$ yields `p(1)` as an answer for `?- p(X).` while $P2$ yields `p(2)`. The opposite happens with the `last` mode. The next programs illustrate the problem of the `sum` mode:

```
% P3
:- table p(sum).
p(1).
```

```
% P4
:- table p(sum).
p(1). p(1).
```

Again the least fixed point semantics of both programs coincides: $lfp(\widehat{T}_{P3}) = \{\texttt{p(1)}\} = lfp(\widehat{T}_{P4})$. However, they produce the following results in Yap:

```
% P3
?- p(X).
X = 1.
```

```
% P4
?- p(X).
X = 1 ; X = 1.
?- p(X).
X = 2.
```

Yap produces the result `p(1)` twice the first time the query is called. Any subsequent query is answered with `p(2)`. In other words, not only are the results of $P3$ and $P4$ not consistent, the results for $P4$ are not internally consistent either.

In the rest of this chapter I disregard these three modes. As their implementations are so obviously sensitive to the program structure, I do not see a good way to reconcile them with my semantics-oriented post-processing specification for aggregation. In fact, in my opinion these modes are best avoided in declarative high-level logic programs.

3.2.2 Lattice-Based Approaches

The remaining three modes, `min`, `max` and `all`, share one notable property: they are all based on a join-semilattice structure defined on (subsets of) U_P, the set of all ground terms over the alphabet of P. A *join-semilattice* is a poset $\langle S, \leq_S \rangle$ such that every finite subset $X \subseteq S$ has a least upper bound in S, denoted by, as in the case of complete lattices, $\bigvee X$. For example, the set of natural numbers with standard order $\langle \mathbb{N}, \leq \rangle$ is a join-semilattice (with $\bigvee X = \mathsf{max}\, X$), but it is not a complete lattice (any infinite subset cannot have an upper bound). Different modes define the following join-semilattices:

- `min` defines the join-semilattice $\langle U_P, \leq \rangle$ where $\leq$ is the lexicographical ordering on terms (`=</2`). The least upper bound is the minimum.
- `max` defines the join-semilattice $\langle U_P, \geq \rangle$ where $\geq$ is the inverse of $\leq$. The least upper bound is the maximum.

- `all` defines the join-semilattice $\langle \mathcal{P}^{\text{fin}}(U_P), \subseteq \rangle$ where $\mathcal{P}^{\text{fin}}(U_P)$ is the set of all finite subsets of U_P. Existing implementations represent sets as lists of terms, which are themselves terms.

The two additional aggregation approaches, offered by XSB, are also based on join-semilattices:

- XSB generalises the above modes to user-defined join-semilattices with the `lattice(`$\bigvee$`/3)` mode that is parametrised in a binary join operator. For instance, the `min` mode can be defined as `lattice(min/3)`.
- XSB also provides a second user-definable mode `po(`$\preceq$`/2)` in terms of a partial order $\preceq$ on U_P. This partial order induces a join-semilattice $\langle \mathcal{P}^{\text{fin}}(U_P), \sqsubseteq \rangle$ where $X \sqsubseteq Y \equiv \forall x \in X : \exists y \in Y : x \preceq y$.

Therefore, in what follows, we only have to deal with lattices that are essentially subsets of U_P, considerably simplifying the formulae.

As we can always reorder arguments and combine multiple join-semilattices into their product join-semilattice, we may assume, without loss of generality, that only the final argument of a predicate is an output tabling mode. That is, all ground atoms have the shape $Q(\mathbf{X}, x)$ where Q is the name of some predicate, $\mathbf{X}$ is a vector of input arguments $X_1, X_2, \ldots, X_n$ and x is the value of the output parameter. I make the simplifying assumption that all predicates are tabled. If a predicate has only input arguments (like tabling without answer subsumption), a (constant) dummy output can always be added.

Mode-directed tabling groups atoms for a predicate Q by distinct values for the *input* arguments $\mathbf{X}$ and aggregates the values of the *output* argument x into a single term. Therefore, a table of aggregated answers is modelled by a function $\mathit{table} : I_P \times U_P^n \to U_P^\perp$ (where I_P is the set of predicate names in P) that maps a pair of a predicate name and inputs to a single aggregated output. The set of aggregate answers $U_P^\perp = U_P \cup \{\perp\}$ is the set of all terms, on which a special element $\perp$ is grafted, to indicate the lack of an answer. This element extends the chosen order on terms $\preccurlyeq$ such that $\perp$ is an (adjoined) bottom element, that is, $\forall x \in U_P^\perp : \perp \preccurlyeq x$. For legibility, I will also sometimes refer to $I_P \times U_P^n \to U_P^\perp$ by $L(\preccurlyeq)$. The lattice structure $\langle U_P^\perp, \preccurlyeq \rangle$ induces a join-semilattice structure $\langle L(\preccurlyeq), \preceq \rangle$ on the set of tables, where $\preceq$ is the pointwise order:

$$f \preceq g \iff \forall (p, \mathbf{X}) \in I_P \times U_P^n : f(p, \mathbf{X}) \preccurlyeq g(p, \mathbf{X})$$

This lattice structure allows us to aggregate over multiple tables, by aggregating the answers pointwise:

$$(\bigvee F)(p, \mathbf{X}) = \bigvee_{f \in F} f(p, \mathbf{X})$$

By storing each individual element of $lfp(T_P)$ into a table and then aggregating over tables we obtain the semantics for mode directed tabling:

Definition 7 (Table Model)**.** Let $\eta : H_P \to (I_P \times U_P^n \to U_P^\perp)$ and $\rho : (I_P \times U_P^n \to U_P^\perp) \to \mathcal{P}(H_P)$ be defined as:

$$\eta(p(\mathbf{X}, x))(q, \mathbf{Y}) = \begin{cases} x & \text{if } p = q \wedge \mathbf{X} = \mathbf{Y} \\ \bot & \text{otherwise} \end{cases}$$

$$\rho(f) = \{p(\mathbf{X}, f(p, \mathbf{X})) \mid f(p, \mathbf{X}) \neq \bot\}$$

Thus, the function η turns an atom into a singleton table, and ρ maps a table to the set of its true atoms. For convenience, η is overloaded, such that when η is applied to a set it means that the regular η is applied elementwise to every element of the set.

To compute the set of all true atoms of a program P, we need to consider the consequence of joining two elements of a semi-lattice in addition to regular logical consequences. This is because, for arbitrary lattices, the result of a join can be distinct from any of its arguments, and thus produce new facts. Thus, we need a new immediate consequence operator $\widehat{T}_P$, which extends the regular T_P operator with answers obtained by joins. Formally, $\widehat{T}_P : \mathcal{P}(H_P) \to \mathcal{P}(H_P)$ is defined as follows, where $\mathcal{P}^{\text{fin}}(A)$ denotes the set of all finite subsets of a set A:

Definition 8 (Extended Immediate Consequence operator)**.**

$$\widehat{T}_P(X) = \bigcup \left\{ \rho(\bigvee Y) \;\middle|\; Y \in \mathcal{P}^{\text{fin}}(\eta(T_P(X))) \right\} \tag{3.1}$$

One can show that $\widehat{T}_P$ is continuous, hence monotone. In fact, for linear orders (such as `min` and `max`), $\widehat{T}_P$ behaves exactly like T_P. Again, think of the least fixed point of $\widehat{T}_P$ as the set of all the answers that can be obtained by the logical rules and the "lattice rules".

The next step is to discard the subsumed answers by applying the join operator on the set of answers. Thus, the set of all true atoms of the program P using mode-directed tabling is given by:

$$\rho\left(\bigvee_{x \in lfp(\widehat{T}_P)} \eta(x)\right) \tag{3.2}$$

Obviously, when $lfp(\widehat{T}_P)$ is infinite, the least upper bound above does not necessarily exist. That is why to give a full denotational semantics of answer subsumption in the next subsection, tables are modelled more abstractly as

complete lattices. Now, to provide some intuition, let us consider an example in which the least upper bound exists.

Example 9. Consider the example from the beginning of the chapter, rewritten using XSB's lattice answer subsumption for the sake of variety:

```
:- table p(lattice(_,_,min/3)).
:- table e/3.
p(X,Y,1) :- e(X,Y,nt).
p(X,Y,D) :- p(X,Z,D1), p(Z,Y,D2), D is D1 + D2
e(a,b,nt). e(b,c,nt). e(a,c,nt).
min(X,Y,Z) :- Z is min(X,Y).
```

Note that the example also tables `e/3` and adds a dummy output parameter (`nt` stands for not tabled), as described above. The don't cares (`_`) in the tabling directive indicate that they are not part of the lattice. In Yap's terminology: they use the `index` tabling mode. The least fixed point semantics of this program, that is $lfp(\widehat{T}_P)$, is given by the following set:

$$\{\texttt{e(a,b,nt)}, \texttt{e(b,c,nt)}, \texttt{e(a,c,nt)}, \texttt{p(a,b,1)}, \texttt{p(b,c,1)}, \texttt{p(a,c,1)}, \texttt{p(a,c,2)}\}$$

The complete lattice on the final argument of `p` is $\langle \mathbb{N}, \geq \rangle$, the reversed standard order. The least upper bound in this lattice is the usual *infimum* on natural numbers.

Interpreted by this lattice, the semantics is

$$\begin{aligned}
&\rho\left(\bigvee_{x \in lfp(\widehat{T}_P)} \eta(x)\right) \\
&= \rho(\bigvee\{\eta(\texttt{e(a,b,nt)}), \eta(\texttt{e(b,c,nt)}), \eta(\texttt{e(a,c,nt)}), \\
&\qquad \eta(\texttt{p(a,b,1)}), \eta(\texttt{p(b,c,1)}), \eta(\texttt{p(a,c,1)}), \eta(\texttt{p(a,c,2)})\}) \\
&= \rho(t) \text{ where } t(q,x,y) = \begin{cases} \texttt{nt} & \text{if } q = \texttt{e} \wedge (x,y) \in \{(\texttt{a},\texttt{b}), (\texttt{a},\texttt{c}), (\texttt{b},\texttt{b})\} \\ 1 & \text{if } (x,y) \in \{(\texttt{a},\texttt{b}), (\texttt{b},\texttt{c})\} \\ \min\{1,2\} & \text{if } (x,y) = (\texttt{a},\texttt{c}) \\ \bot & \text{otherwise} \end{cases} \\
&= \{\texttt{e(a,b,nt)}, \texttt{e(b,c,nt)}, \texttt{e(a,c,nt)}, \texttt{p(a,b,1)}, \texttt{p(b,c,1)}, \texttt{p(a,c,1)}\}
\end{aligned}$$

Only the atoms representing the shortest paths are retained, as expected.

Example 10. This example illustrates why we need to extend the T_P operator to include the results of the lattice operations, that is, why we need the $\widehat{T}_P$

operator. Consider the lattice $\{a, b, c, d\}$, with $a, b \leq c$ and $c \leq d$, which is used in the following program:

```
lub(a,b,c). lub(a,c,c). lub(a,d,d).
lub(b,a,c). lub(b,c,c). lub(b,d,d).
lub(c,d,d).
lub(X,X,X).

:- table p(lattice(lub/3)).

p(a).
p(b).
p(d) :- p(c).
```

The regular immediate consequence gives us $\mathit{lfp}(T_P) = \{\texttt{p(a)}, \texttt{p(b)}\}$, which means that $\rho(\bigvee \eta(\mathit{lfp}(T_P))) = \{\texttt{p(c)}\}$. The atom `p(c)` does not follow from the logical inference, but from the lattice's join operator. It is included in the overall answer thanks to the post-processing step, but its logical consequences are not. With the $\widehat{T}_P$ operator we have $\mathit{lfp}(\widehat{T}_P) = \{\texttt{p(a)}, \texttt{p(b)}, \texttt{p(c)}, \texttt{p(d)}\}$, and so $\rho(\bigvee \eta(\mathit{lfp}(\widehat{T}_P))) = \{\texttt{p(d)}\}$, which is the intended semantics.

3.2.3 Answer Subsumption for Complete Lattices

Even though at any point of computation each table is finite, it is potentially infinite when a program produces infinitely many answers. Thus, to give a denotational semantics for answer subsumption, a join-semilattice on terms is not enough, as we need least upper bounds of infinite sets, i.e. a complete lattice structure. For example, the most natural candidate for the types of values in the case of the `all` mode is $\langle \mathcal{P}(U_P), \subseteq \rangle$, the complete lattice of all subsets of U_P, which cannot be modelled by (finite) terms. In general, every semilattice can be extended to a complete lattice via MacNeille (1937) completion.

Thus, no longer imposing any order on the set $U_P^{\perp}$, the type of the table becomes $(I_P \times U_P^n \to L)$ for a complete lattice L. Each predicate $p \in I_P$ has bottom-preserving *abstraction* and *representation* functions: $\lceil \text{-} \rceil_p : U_P^{\perp} \to L$ and $\lfloor \text{-} \rfloor_p : L \to U_P^{\perp}$ respectively. Representation $\lceil \text{-} \rceil_p$ is required to be a retraction of $\lfloor \text{-} \rfloor_p$, that is, $\lceil \lfloor x \rfloor_p \rceil_p = x$. Since the two functions must preserve bottoms, the least element of L denotes "no value". This, gives new definitions of η and ρ, appropriately adding abstraction and representation, where $\perp^L$ is the least element of L:

Definition 11 (Table Model for Complete Lattices)**.**

$$\eta(p(\mathbf{X}, x))(q, \mathbf{Y}) = \begin{cases} \lceil x \rceil_p & \text{if } p = q \wedge \mathbf{X} = \mathbf{Y} \\ \bot^L & \text{otherwise} \end{cases}$$

$$\rho(f) = \left\{ p(\mathbf{X}, \lfloor f(p, \mathbf{X}) \rfloor_p) \mid f(p, \mathbf{X}) \neq \bot^L \right\}$$

To give the semantics, the $\widehat{T}_P$ operator is defined exactly as in (3.1) but using the new definitions of η and ρ. It is easy to see that it is monotone, so it always has a least fixed point. The semantics of the entire program is given again as in (3.2).

3.2.4 Lattice Semantics for Stratified Programs

General programs are stratified to distinguish between predicates that imply and are implied by tabled values. The *depends on* relation $\ltimes$ is defined as follows: for any two predicates `p` and `q`, it is the case that `p` $\ltimes$ `q` if and only if there exists a clause `p(...):-...,q(...),...`. We say that `p` and `q` are in the same stratum if `p` $\ltimes^+$ `q` and `q` $\ltimes^+$ `p`, where $\ltimes^+$ is the reflexive and transitive closure of $\ltimes$. Put differently, a stratum is a strongly connected component of the dependency graph defined by $\ltimes$. The relation $\ltimes$ induces a partial ordering on the set of all strata S, that is, for $X, Y \in S$, it is the case that $X \leq Y$ if and only if there exists `p` $\in X$ and `q` $\in Y$ such that `p` $\ltimes^+$ `q`.

A stratum forms a logical unit to which the least fixed point semantics and aggregation are applied in turn: For each stratum $X \in S$, its semantics $M_X \subseteq \mathcal{P}(H_P)$ is defined as follows:

$$M_X = \rho(\bigvee_{x \in lfp(\widehat{T}_{Q_X})} \eta(x)) \qquad \text{given} \qquad Q_X = P_X \cup \bigcup_{Y<X} M_Y,$$

where P_X is $\mathsf{ground}(P)$ restricted to the predicates in the stratum X, and $\bigcup_{Y<X} M_Y$ should be understood as a set of facts. Informally, this means that to give a semantics for a stratum, (1) first compute the semantics of the strata below, (2) use the results as a set of facts added to the part of the program in the current stratum, (3) compute the fixed point, and (4) finally perform the aggregation step using the join operator. There are always finitely many strata, so M_X is well-defined. The semantics of the program P is then the aggregation of the sum of the interpretations of all the strata, that is, $\bigcup_{X \in S} M_X$.

Stratification ensures that the answers for a predicate are always aggregated before they are used by another predicate, unless there is a cyclic dependency

between them. for example, consider the following variation on the shortest path program:

```
:- table p(index,index,min).
e(1,2). e(2,3). e(1,3).
p(X,Y,1) :- e(X,Y)
p(X,Y,D) :- p(X,Z,D1),p(Z,Y,D2), D is D1 + D2.
s(X,Y,D) :- p(X,Y,D).
```

Without stratification, the semantics is given by $\rho(\bigvee \eta(\mathit{lfp}(\widehat{T}_P))$ which contains `s(1,3,2)`, as a consequence of `p(1,3,2)` before aggregation. However, if we stratify the program as discussed above, the rules for `p` end up in a lower stratum than `s`. Therefore, the results for `p` will be aggregated by `min`, before any consequence is derived from it. Since `p(1,3,2)` is subsumed, `s(1,3,2)` is no longer derived.

When two predicates are interdependent (and therefore in the same stratum), but only one of them is tabled, the answers for the untabled predicate are not subsumed. Stratification then gives a result that appears inconsistent:

```
:- table even(min).

even(0).
even(X) :- odd(Y), Y is X - 1.
odd(X) :- even(Y), Y is X - 1.
```

The semantics interprets this program as the set

$$\rho(\bigvee \eta(\mathit{lfp}(\widehat{T}_P))) = \{\texttt{even(0)}, \texttt{odd(1)}, \texttt{odd(3)}, \texttt{odd(5)}, \texttt{odd(7)}, \ldots\}.$$

While $\{\texttt{even(0)}, \texttt{odd(1)}\}$ seems equally reasonable. Because subsumption is treated as a post-processing step *per stratum*, inter-dependent predicates are resolved as if no answers were subsumed. Subsumption only affects predicates in the strata above. For instance, assume we add the following (non-tabled) predicate to the program:

```
also_odd(X) :-  even(Y), Y is X - 1.
```

It is in a different stratum than `even` and `odd`, so its semantics depends on the semantics of `even` after the subsumption step. This means that the semantics together with the `also_odd` predicate is given as:

$$\{\texttt{even(0)}, \texttt{also_odd(1)}, \texttt{odd(1)}, \texttt{odd(3)}, \texttt{odd(5)}, \texttt{odd(7)}, \ldots\},$$

Here `also_odd` behaves like the alternative suggested for `odd` above. Importantly, programs like the one above do not satisfy the correctness condition for the greedy strategy given in Section 3.3.

3.3 Generalised Answer Subsumption Semantics

The previous section specifies the semantics of tabling with lattice-based answer subsumption in terms of a post-processing aggregation. However, the existing implementations do not actually first compute the least Herbrand model. Instead, they greedily aggregate intermediate results during SLD-resolution. This makes it feasible to, for instance, compute the shortest path in a cyclic graph in a finite amount of time, as well as generally improving efficiency. However, as the examples at the start of the chapter acutely demonstrate, this greedy strategy is not always valid. This section characterises the greedy strategy as a form of generalised semantics for logic programs and considers its correctness with respect to the postulated specification.

3.3.1 Generalised Semantics

Again, assume that we work in a single stratum. We can capture greedy strategies as *generalised semantics* of P in terms of a generalised model:

Definition 12 (Generalised Model)**.** A generalised model is a 3-tuple $\langle L, \eta^L, T_P^L \rangle$, where:

1. L is a complete lattice,
2. $\eta^L : H_P \to L$ is a function that "embeds" terms in the lattice,
3. $T_P^L : L \to L$ (a *generalised* immediate consequence operator) is monotone,
4. L is generated by $\eta^L(H_P)$, which means that for every $x \in L$, there exists an $X \subseteq \eta^L(H_P)$ such that $x = \bigvee X$.

Note that in general we do not need a counterpart of the ρ function: in the post-processing semantics given in Section 3.2.2, we need ρ to define the $\widehat{T}_P$ operator and move between strata. Here, we use this semantics to capture correctness within a single stratum, and the immediate consequence operator is not defined, but it is given. This allows us to generalise the whole table to a lattice, which simplifies the generalised semantics.

The generalised semantics of the program P is given by $\mathit{lfp}(T_P^L)$, which exists due to the Knaster–Tarski theorem. It is easy to verify that the regular least fixed point semantics is a valid instance of this generalised semantics. Also, note that the generalised semantics does not depend on η^L or the fact that L is generated by η^L. We need these facts in a moment to establish a correctness criterion.

One obvious instantiation of this semantics is with the $\widehat{T}_P$ operator defined in Section 3.2.3, where $\iota : X \to \mathcal{P}(X)$ is the singleton function:

$$\langle \mathcal{P}(H_P), \iota, \widehat{T}_P \rangle$$

We say that an instance of the generalised semantics is a correct implementation strategy only if yields the same result as the post-processing semantics defined in Section 3.2.2. More formally, a generalised semantics $\langle L, \eta^L, T_P^L \rangle$ is *correct* if and only if

$$\mathit{lfp}(T_P^L) = [\eta^L](\mathit{lfp}(\widehat{T}_P)), \tag{3.3}$$

where for every lattice L, set S, and function $f : S \to L$, the function $[f] : \mathcal{P}(S) \to L$ is defined as:

$$[f](X) = \bigvee_{x \in X} f(x)$$

Intuitively, equation (3.3) means that evaluating the whole program with no answer subsumption and then obtaining the final result by post-processing with L's join operation is the same as computing every step with the lattice L (which is usually more efficient). The following theorem gives sufficient conditions for equation (3.3) to hold.

Theorem 13 (Fixed Point Fusion (Backhouse, 2000))**.** *Let $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ be posets. Let $f : X \to X$ and $g : Y \to Y$ be two functions with least fixed points. Let $h : X \to Y$ be a function. Assume the following:*

a. *It is the case that $h \circ f = g \circ h$.*

b. *The function h has an upper Galois adjoint, that is, there exists a function $j : Y \to X$ such that $h(x) \leq_Y y \iff x \leq_X j(y)$ for all $x \in X$ and $y \in Y$.*

Then, $\mathit{lfp}(g) = h(\mathit{lfp}(f))$.

Equation (3.3) is clearly an instance of this theorem's conclusion, with $f = \widehat{T}_P$, $g = T_P^L$ and $h = [\eta^L]$. Yet, the theorem only applies if we can show that (a) $[\eta^L] \circ \widehat{T}_P = T_P^L \circ [\eta^L]$, and that (b) $[\eta^L]$ has an upper Galois adjoint.

Fortunately, (b) readily follows from the fact that $[\eta^L]$ is continuous, and that every continuous function has an upper Galois adjoint (Backhouse, 2000).

Intuitively the remaining condition (a) means that $[\eta^L]$ should preserve immediate consequences. In other words, subsumption of a conventional immediate consequence should be the generalised immediate consequence of $[\eta^L]$.

3.3.2 Greedy Strategy

For better performance, practical implementations of tabling with answer subsumption use a greedy strategy, which means that they remove subsumed answers as soon as possible, and not as a single post-processing step. We can try to express it as the following instance of the generalised semantics, in terms of the functions defined in Section 3.2.3:

$$\langle I_P \times U_P^n \to L, \eta, [\eta] \circ T_P \circ \rho \rangle$$

The function ρ extracts the set of true ground atoms from a table, which is a subset of the Herbrand base H_P, so we can indeed express the 'greedy' immediate consequence operator in terms of the standard immediate consequence operator T_P, subsumption $[\eta]$ and extraction ρ. In fact this definition is quite convenient, because tabled Prolog systems readily provide an efficient implementation of T_P.

The question remains if the function $[\eta] \circ T_P \circ \rho$ has a fixed point. Luckily, given a tuple $\langle L, \eta^L, T_P^L \rangle$, where L is generated by $\eta^L(H_P)$, the condition (a) is enough for T_P^L to be monotone (this is where the assumption that $\eta^L(H_P)$ generates L comes in handy):

Theorem 14. *Let $\langle L, \eta^L, T_P^L \rangle$ be as above. If $[\eta^L] \circ \widehat{T}_P = T_P^L \circ [\eta^L]$ (the condition (a)), then T_P^L is monotone.*

This theorem is proved at the end of this chapter.

Additionally, it is the case that $[\eta] \circ T_P = [\eta] \circ \widehat{T}_P$. Thus, to show correctness of a program under the greedy semantics, we need only show the following:

$$[\eta] \circ T_P = [\eta] \circ T_P \circ \rho \circ [\eta] \tag{3.4}$$

Example 15. Reconsider the invalid program from the beginning of this chapter. We show that the condition 3.4 does not hold for this program under semantics $\langle L(\preccurlyeq), \preceq, \eta, T_P^L(\preccurlyeq) \rangle$ where $H_P = \{\texttt{p(0)}, \texttt{p(1)}, \texttt{p(2)}, \texttt{p(3)}\}$ and $(\preccurlyeq) = (\leq)$ is the partial order on terms. By means of the following counter example: A simple calculation shows that $([\eta] \circ T_P)\big(\{\texttt{p(0)}, \texttt{p(1)}\}\big) \neq ([\eta] \circ T_P \circ \rho \circ [\eta])\big(\{\texttt{p(0)}, \texttt{p(1)}\}\big)\big)$.

This example also explains the odd behaviour of XSB, Yap and B-Prolog: `p(0)` is subsumed by `p(1)`, therefore the body of the third rule in the program cannot be satisfied and `p(3)` (the correct answer) is never produced.

Interestingly, when we change the last rule to `p(3) :- p(0)`, the result of the query `?-p(X).` changes once more in all systems, although logically both rules should behave identically. Furthermore, different implementations handle this rule differently. For instance, XSB reasonably disallows calls where `lattice`-mode arguments are not free, and the latter rule therefore produces an error message. While Yap instead produces `X=3`, because its batched-mode evaluation immediately derives `p(1)` from the fact `p(1)`.

Example 16. Now, consider the shortest path program from Example 9 under semantics $\langle L(\preccurlyeq), \preceq, \eta, T_P^L(\preccurlyeq)\rangle$ and $(\preccurlyeq) = (\geq)$. We prove the correctness condition by proving that $lhs \preceq rhs$ and $lhs \succeq rhs$. Then by anti-symmetry of $\preceq$, the correctness condition holds.

($\succeq$-direction) Because $[\eta] \circ T_P$ is monotone, and for this case $\rho \circ [\eta]$ is deflative, i.e. $\rho([\eta](X)) \subseteq X$ for all $X \subseteq H_P$, we are done with this direction.

($\preceq$-direction) A `p(x,y,d)`-atom is either introduced by an edge `e(x,y,nt)` or by two other atoms `p(x,z,d1)` and `p(z,y,d2)`. In the former case $d = 1$, in the latter case $d = d_1 + d_2$. It is easy to see that if an atom `p(x,y,1)` is introduced by T_P, it is also introduced by $T_P \circ \rho \circ [\eta]$. In the latter case, d_1 and d_2 are at least as large as their corresponding entries in the table produced by $[\eta]$. Hence, d must be at least as large as any corresponding distance in the set produced by $T_P \circ \rho \circ [\eta]$. Hence, the infimum of ds produced by T_P must be at least as large as any corresponding distance. Finally, this means that the entry in the table produced by $[\eta] \circ T_P$ must be at least as large as the entry in the table produced by $[\eta] \circ T_P \circ \rho \circ [\eta]$.

3.4 Historical Notes and Related Work

As far as I know, "output" subsumption for tabling was first proposed by Van Hentenryck et al. (1993) in the context of abstract interpretation.

Nowadays, answer subsumption seems to be used most often for dynamic programming. In dynamic programming, an optimal solution to a problem is defined in terms of the optimal solutions of smaller sub-problems. This intuition is in fact captured by the correctness condition (Equation 3.4): it states that the solution found by examining all sub-solutions, is equal to examining only the optimal solutions. This is good news, because it implies correctness for dynamic programming.

I have already discussed the tabling modes of Yap (Santos and Rocha, 2013) and XSB (Swift and Warren, 2012) at length in Section 3.2.1. XSB's lattice based answer subsumption is more suitable for implementing techniques that require more general lattices than simple minimum and maximum, such as abstract interpretation.

Guo and Gupta (2004, 2008) implemented 5 tabling modes in ALS-Prolog with the aim of simplifying and accelerating dynamic programming. These modes are `+` (indexed), `-` (only the first answer is retained), `min` (minimum), `max` (maximum) and `last` (the last answer is retained). The correspondence to Yap's tabling modes is obvious, but there is no equivalent for Yap's `all` and `sum` modes. Answers are grouped by distinct values for the `+` arguments and the remaining arguments are aggregated based on their mode.

Zhou et al. (2010) added tabling to B-Prolog with the same purpose in mind. Consequently, the tabling modes they support are identical, except they do not implement a `last` mode. Instead, they support cardinality constraints, which limit the answers that are stored in the table to the first N optimal answers for some positive integer N. Also supported is an `nt` (not-tabled) tabling mode, which is used to pass around global constants efficiently. From the perspective of the tabling system `nt` arguments do not exist, and thus are never stored in the table.

Recently, SWI-Prolog (Wielemaker et al., 2012) has been extended with support for tabling, including answer subsumption. SWI-Prolog's tabling support was implemented using a novel technique (Desouter et al., 2015), made possible by SWI's extensions for delimited continuations (Schrijvers et al., 2013). SWI implements lattice, partial order, `first`, `last`, `min`, `max`, and `sum` tabling modes.

The tabling mechanism of SWI-Prolog has been further extended with answer subsumption. This new functionality has been used to implement a probabilistic logic programming system, PITA, in SWI-Prolog Riguzzi et al. (2018). The implementation hooks into the answer subsumption system to transform the Prolog program into Binary Descision Diagrams (BDDs). BDDs are a compact data structure that enables efficient inference of probabilistic queries (De Raedt et al., 2007).

Tabled Constraint Logic Programming Constraint Logic Programming extends the logic programming paradigm with constraints, i.e., equalities or inequalities, usually involving variables on a certain domain (e.g., integers). Arias and Carro (2016) describe a universal approach for combining tabling and CLP (TCLP). In subsequent work (Arias and Carro, 2019), they define a program transformation from Prolog with answer subsumption to TCLP. In

this transformation, every predicate automatically implies all predicates that it subsumes (and transitively, their consequences).

Unfortunately, their work does not provide a formal treatment of the semantics of this transformation. Briefly, it seems that we can capture their semantics as a generalised semantics $\langle I \times U_P \to U_P^{\perp}, \eta, [\eta] \circ T_P \circ \rho_{\uparrow} \rangle$ where:

$$\rho_{\uparrow}(f) = \{p(\mathbf{X}, \mathbf{Y}) \mid \mathbf{Y} \in U_P^{\perp}, \perp \neq \mathbf{Y} \preccurlyeq f(p, \mathbf{X})\}$$

It can be shown that for all programs P and for all $S \subseteq H_P$:

$$([\eta] \circ T_P)(S) \preceq ([\eta] \circ T_P \circ \rho_{\uparrow} \circ [\eta])(S)$$

A variant of Fixed-Point Fusion (Backhouse, 2000, p.129) then tells us that:

$$[\eta](\mathit{lfp}(\widehat{T_P})) \preceq \mathit{lfp}([\eta] \circ T_P \circ \rho_{\uparrow})$$

That is, the TCLP system always computes an *over-approximation*: it may contain solutions that are too good according to the denotational semantics. In contrast, the greedy semantics computes an under-approximation: it may miss out on certain more optimal solutions.

As example of where such an over-approximation may be problematic, consider the program:

```
:- table p(max).
p(1).
p(2) :- p(0).
```

In this case answer subsumption returns `p(1)`, whereas TCLP returns `p(2)`, because `p(0)` is implied by `p(1)`.

Semi-naive evaluation Semi-naive evaluation is a bottom-up strategy (whereas tabling is top-down) for computing least fixed points of Datalog (Ceri et al., 1989) programs. This method is efficient when it can incrementally computes new solutions, and avoids recomputing previously discovered solutions. Arntzenius and Krishnaswami (2020) define a program transformation for their typed functional language Datafun (Arntzenius and Krishnaswami, 2016), which accomplishes this semi-naive evaluation. The type system only admits monotone functions on (finite) lattices, meaning that functions in Datafun always have a least fixed point. Potentially, a similar type system could be devised for Prolog to only admit programs that satisfy the correctness condition (Equation (3.4)), e.g., an adaption of Schrijvers et al. (2008)'s type system for Prolog.

3.5 Proof of Monotonicity of T_P^L

The following is a more independent statement of Theorem 14:

Theorem (Theorem 14)**.** *Let L be a complete lattice, $\eta^L : H_P \to L$ and $T_P^L : L \to L$ functions such that L is generated by $\eta^L(H_P)$ and $[\eta^L] \circ \widehat{T}_P = T_P^L \circ [\eta^L]$, then T_P^L is monotone.*

Proof. Let $x \leq_L y$. Since L is generated by $\eta^L(H_P)$, there exist $X \subseteq Y \subseteq H_P$ such that $x = \bigvee \eta^L(X)$ and $y = \bigvee \eta^L(Y)$.

$$
\begin{array}{rcll}
T_P^L(x) & = & T_P^L(\bigvee \eta^L(X)) & \\
& = & T_P^L([\eta^L](X)) & \text{(def. of } [\eta^L]) \\
& = & [\eta^L](\widehat{T}_P(X)) & \text{(assumption)} \\
& \leq_L & [\eta^L](\widehat{T}_P(Y)) & \text{(composition preserves monotonicity)} \\
& = & T_P^L([\eta^L](Y)) & \text{(assumption)} \\
& = & T_P^L(\bigvee \eta^L(Y)) & \text{(def. of } [\eta^L]) \\
& = & T_P^L(y) &
\end{array}
$$

□

3.6 Scientific output

Table 3.1 summarises the various semantics that have been defined in this chapter.

This chapter draws material from:

> Alexander Vandenbroucke, Maciej Piróg, Benoit Desouter, and Tom Schrijvers. 2016. Tabling with Sound Answer Subsumption. In Theory and Practice of Logic Programming. Cambridge University Press, 2016, Vol. 16, iss. 5, pp. 933–949. DOI: `http://doi.org/10.1017/S147106841600048X`

	Table Model	Immediate Consequence	Semantics
no subsumption	N/A	T_P	$lfp(T_P)$
semi-lattice	Definition 7	$\widehat{T}_P$	$\rho(\bigvee_{x \in lfp(\widehat{T}_P)} \eta(x))$
complete lattice	Definition 11	$\widehat{T}_P$	$\rho(\bigvee_{x \in lfp(\widehat{T}_P)} \eta(x))$
stratified	Definition 11	$\widehat{T}_{Q_X}$ (for each $X \in S$)	$\bigcup_{X \in S} M_X$
generalised	N/A	T_P^L	$lfp(T_P^L)$

Table 3.1: Summary of the semantics of tabled programs.

This is a multi-author work, where the contribution of each author is the following: Most of the issues discussed in Section 3.1 were initially pointed out by Benoit Desouter. The subsequent theoretical developments were accomplished jointly by the other authors. The material in Section 3.2 is mostly due to the author of this thesis, whereas the material in Section 3.3 is mostly due to Maciej Piróg. In particular, Definition 12, Theorem 14 and Equation 3.4 are mostly his contribution.

Chapter 4

Monadic Tabling in Haskell

Non-determinism models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. Unlike Prolog, Haskell does not natively have non-determinism. Instead, the conventional approach (Wadler, 1985) represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

```
1 ? 2
```

This expression represents a non-deterministic choice (with the operator `?`) between `1` and `2`. Traditionally we model this with the list `[1,2]`.

Now consider the next example:

```
swap (m,n) = (n,m)

pair = (1,2) ? swap pair
```

The corresponding Haskell code also appeared in Chapter 2:

```
p :: [(Int,Int)]
p = [(1,2)] ++ map swap p where swap (x,y) = (y,x)
```

However, this model is flawed, as its execution shows:

```
> p
[(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),...
```

That is, we get an infinite list, although only two distinct outcomes exist. The situation is identical to standard Prolog versus tabled Prolog: Prolog's least fixed point semantics clearly only admits (sets of) distinct solutions, yet practical implementations can return the same answer multiple times. This suggests that non-determinism should be modelled as a *set* instead of a list: The expected model of `p` is the set `{(1,2),(2,1)}`:[1]

```
p :: Set (Int,Int)
p = Set.singleton (1,2) `Set.union` Set.map swap p where
  swap (x,y) = (y,x)

> p
fromList *** Exception: <loop>
```

Haskell lazily prints the first part of the `Set` constructor, and then has to compute the union infinitely many times. As an executable model of it clearly remains inadequate: it fails to compute the solution `{(1,2),(2,1)}`.

Semantically, the reason is that Haskell's recursion does not compute a fixed point in the powerset lattice (as is the a case for Prolog's least fixed point semantics), but rather in some other lattice (technically a complete partial order). In this lattice, the least fixed point is $\bot$, a value representing non-terminating recursion (Danielsson et al., 2006). This is clearly not the desired solution.

In short, we cannot rely upon Haskell's native recursion to compute the least fixed point for us. Instead, we must compute the least fixed point ourselves by iterating, essentially implementing a tabling mechanism in Haskell.

The first half of the implementation consists of a monadic model that captures both non-determinism *and* recursion, yielding a finite representation of recursive non-deterministic expressions. This representation can be used as a light-weight embedded domain specific language to build non-deterministic expressions in Haskell.

The other half of the implementation is a denotational semantics in terms of the least fixed point of a function $\mathcal{R}[\![\cdot]\!]$. This semantics is subsequently implemented as a Haskell function `runTable` that interprets the model.

The denotational semantics is then generalised to arbitrary complete lattices. This gives additional expressive power to the tabling model (similar to answer subsumption). The added power is demonstrated on a simple graph problem—finding the shortest path in a cyclic graph, which could not be solved in a finite amount of time with the earlier semantics.

[1] `Set` is defined in the `Data.Set` module of the `containers` package.

4.1 Non-determinism

A computation is non-deterministic if it has several possible outcomes. This chapter models such non-deterministic computations as the set of their results.

Consider the following non-deterministic expression that produces either `1` or `2`:

$$1 \ ? \ 2$$

For this example, the semantics is given by the set $\{1, 2\}$. The semantic function $[\![\cdot]\!]$ formally characterizes this interpretation:

$$[\![n]\!] = \{n\}$$

$$[\![\texttt{e1 ? e2}]\!] = [\![\texttt{e1}]\!] \cup [\![\texttt{e2}]\!]$$

The semantics of a literal n is the singleton of n, and the semantics of a choice `e1 ? e2` is the union of the semantics of the left and right branches. A simple calculation shows that $[\![\texttt{1 ? 2}]\!]$ is indeed $\{1, 2\}$.

$$[\![\texttt{1 ? 2}]\!] = [\![1]\!] \cup [\![2]\!] = \{1\} \cup \{2\} = \{1, 2\}$$

Let us extend our semantics to allow for addition of non-deterministic values:

$$[\![e_1 + e_2]\!] = \{n + m \mid n \in [\![e_1]\!], m \in [\![e_2]\!]\}$$

Now, consider the expression:

$$(1 \ ? \ 2) + (1 \ ? \ 2)$$

Here we have an expression that contains an addition of two choices. This expression has 4 possible outcomes of which two coincide: the result is either 2 (1+1), 3 (1+2 or 2+1) or 4 (2+2).[2] Again calculation gives the expected result:

$$\begin{aligned}
&[\![\texttt{(1 ? 2) + (1 ? 2)}]\!] \\
&= \{n + m \mid n \in [\![\texttt{1 ? 2}]\!], m \in [\![\texttt{1 ? 2}]\!]\} \\
&= \{n + m \mid n \in \{1, 2\}, m \in \{1, 2\}\} \\
&= \{2, 3, 4\}
\end{aligned}$$

[2] The conventional list-based semantics would be `[2,3,3,4]`.

Recursion The next example presents a recursive non-deterministic expression `pair` which chooses between `(1,2)` and the new primitive `swap` to swap `pair`'s components around.

```
pair = (1,2) ? swap pair
```

The semantics of `swap e` is the set obtained by flipping all pairs in the set of results of `e`:

$$[\![\texttt{swap e}]\!] = \{(m,n) \mid (n,m) \in [\![e]\!]\}$$

The semantics of `pair` are given by the following recursive equation:

$$\begin{aligned} & [\![\texttt{pair}]\!] = [\![\texttt{(1,2) ? swap pair}]\!] \\ \iff & [\![\texttt{pair}]\!] = [\![\texttt{(1,2)}]\!] \cup [\![\texttt{swap pair}]\!] \\ \iff & [\![\texttt{pair}]\!] = \{(1,2)\} \cup \{(m,n) \mid (n,m) \in [\![\texttt{pair}]\!]\} \end{aligned} \tag{4.1}$$

This equation admits infinitely many solutions, e.g.

$$[\![\texttt{pair}]\!] = \{(1,2),(2,1)\},$$

$$[\![\texttt{pair}]\!] = \{(0,0),(1,2),(2,1)\},$$

$$[\![\texttt{pair}]\!] = \{(1,1),(1,2),(2,1)\},$$

$$\ldots$$

However, we can identify a *least* solution: the set $\{(1,2),(2,1)\}$ is contained in every other solution.

As we saw previously, a naive translation of this idea in Haskell *does not work*:

```
  det :: Ord a => a -> Set a
  det x = singleton x

  (?) :: Ord a => Set a -> Set a -> Set a
  a ? b = union a b

  swap :: (Ord a, Ord b) => Set (a,b) -> Set (b,a)
  swap = Data.Set.map (\(x,y) -> (y,x))

  pair :: Set (Int,Int)
  pair = det (1,2) ? swap pair

> pair
fromList *** Exception
```

The reason it does not work is that under Haskell's (cpo-based) semantics the equation (4.1) has a different least solution: $\bot$ (i.e. non-termination).[3] As this additional $\bot$ in the domain is clearly not the desired solution, we cannot rely on Haskell's native semantics for recursion.

The main novelty of this chapter is to reformulate the problem as a *different* least fixed point problem, for which we can iteratively compute the solution. Moreover, our approach incurs minimal overhead for the programmer, compared to writing the function using the conventional recursive approach.

4.2 Effect Handlers

Monads are a way to model side-effects such as non-determinism in a pure functional programming language (Wadler, 1995). This chapter uses effect handlers to construct a monad for non-determinism and recursion. Effect handlers (Kiselyov et al., 2013; Schrijvers et al., 2019) factor the problem of modeling effectful computations into two parts: first a syntax is introduced to represent all relevant operations, second effect handlers are defined that interpret the syntax within a semantic domain.

The syntax of non-deterministic computations can be modeled with the following data type `ND`, which supports three operations: `SuccessND a` is a deterministic computation with result `a`, `Fail` is a failed computation, and `Or l r` represents a non-deterministic choice between two non-deterministic computations `l` and `r`.

```
data ND a
  = SuccessND a
  | FailND
  | OrND (ND a) (ND a)
```

Because the above data type is a free monad (Swierstra, 2008) we can easily define the following `Monad` instance:

```
instance Monad ND where
  return a = SuccessND a
  SuccessND a >>= f = f a
  FailND      >>= f = FailND
  OrND l r    >>= f = OrND (l >>= f) (r >>= f)
```

[3] In Haskell we work in the domain $\langle \mathcal{P}(\mathbb{Z} \times \mathbb{Z}) \cup \{\bot\}, \sqsubseteq \rangle$, where every type is inhabited by $\bot$, representing non-termination, and $\bot \sqsubseteq v$ for any value v.

This monad instance substitutes `f a` for every `SuccessND a` in the data structure, leaves `Fail` untouched, and recurses on both branches of `Or l r`. With this monad instance, the example `(1 ? 2) + (1 ? 2)` is expressed as:

```
exampleND :: ND Int
exampleND = do x <- return 1 `OrND` return 2
               y <- return 1 `OrND` return 2
               return (x + y)
```

Values of type `ND a` are abstract syntax trees. The function `exampleND` constructs such an abstract syntax tree. The interpreter `nd` decodes this tree according to the semantics defined by $[\![\cdot]\!]$. It turns `Success` into a singleton set, `Fail` into an empty set, and `Or` into the union of the interpretations of its branches.

```
nd :: Ord a => ND a -> Set a
nd (SuccessND a) = singleton a
nd FailND        = empty
nd (OrND l r)    = union (nd l) (nd r)

> nd exampleND
fromList [2,3,4]
```

The equivalent of `pair` in the abstract syntax is `pairND`:

```
pairND :: ND (Int, Int)
pairND = return (1,2) `OrND` fmap swap pairND where
  swap (x,y) = (y,x)

> nd pairND
fromList ...
```

Encoding the non-deterministic syntax as a data type does not solve the problem of non-termination. This can be explained by looking at the syntax tree for `pairND` in Figure 4.1a: it is an infinite tree with an `Or` node at the root, a `Success` node in the left branch, and in the right branch another tree with `Or` at the root after which the pattern repeats. Obviously the interpreter `nd` does not terminate when interpreting such an infinite tree.

We have little chance of processing the infinite tree in a finite time if we do not represent it in a finite way. Yet the current recursive form of `pairND` hides the recursive call in the function body, making the construction of an infinite tree unavoidable.

4.3 Explicating Recursion

In order to obtain a finite syntax tree, we have to once more change the representation of non-deterministic computations. First, we add a constructor to the abstract syntax to explicitly represent a recursive call. Second, we replace all recursive calls with this constructor, and finally, we define a new effect handler that interprets the now finite syntax tree, producing the desired solution. The following data type models recursive calls in addition to non-determinism:

```
1  data Table i o a
2    = Success a
3    | Fail
4    | Or (Table i o a) (Table i o a)
5    | Rec i (o -> Table i o a)
```

The first three constructors (lines 2–5) capture non-determinism, exactly like the previously defined data type `ND`. The last constructor (line 5) captures a recursive call: `Rec a k` represents a recursive call with argument `a :: i`, and continuation `k :: o -> Table i o a`. For convenience, we define four additional smart constructors. The smart constructor `rec` performs a recursive call and immediately wraps the result in a successful computation:

```
rec :: i -> Table i o o
rec i = Rec i Success
```

The smart constructor `choice` picks a computation from a list in a non-deterministic fashion.

```
choice :: [Table i o a] -> Table i o a
choice = foldr Or Fail
```

The smart constructor `choose` picks an element from a list in a non-deterministic fashion.

```
choose :: [a] -> Table i o a
choose = choice . map Success
```

The smart constructor `guard` returns `()` if its argument is true and fails otherwise.

```
guard :: Bool -> Table i o ()
guard b = if b then return () else Fail
```

The data type `Table` is again a free monad, the corresponding monad instance is:

```
1  instance Monad (Table i o) where
2    return a = Success a
3    Success a >>= f = f a
4    Fail      >>= f = Fail
5    Or l r    >>= f = Or (l >>= f) (r >>= f)
6    Rec i k   >>= f = Rec i (k >=> f)
```

Lines 1–5 are identical to the `Monad` instance for `ND`. Line 6 defines that `>>=` of a recursive call is obtained by a new recursive call to the same argument, but with an extended continuation. [4]

As an example, consider this latest incarnation of `pair`, written in the `Table` syntax:

```
pairTable :: () -> Table () (Int,Int) (Int,Int)
pairTable () = return (1,2) `Or` do (x,y) <- rec ()
                                    return (y,x)
```

The next section builds an interpreter for this program.

4.4 Effect Handler for Explicit Recursion

4.4.1 Denotational semantics

This section formalises the meaning of the abstract syntax. The meaning of a non-deterministic function f of type[5] $I \to$ `Table` $I\ O\ O$, mapping I onto `Table` $I\ O\ O$ is given by a function $[\![f]\!] : I \to \mathcal{P}(O)$. This function maps values of type I onto a subset of the values described by the type O.

$$[\![\cdot]\!] : (I \to \texttt{Table}\ I\ O\ O) \to (I \to \mathcal{P}(O))$$

[4] The new continuation is the Kleisli composition `k >=> f` where `(k >=> f) x = k x >>= f`.

[5] In deference to mathematical convention, I use uppercase characters for meta-variables I and O when using mathematical syntax.

Let us first define the semantics of an easier case: suppose we already have an environment $s : I \to \mathcal{P}(O)$ that contains a partial set of solutions for every call $f(a)$, i.e.

$$s(a) \subseteq [\![f]\!](a) \text{ for every } a \in I$$

Then for every syntax tree $t : \texttt{Table}\ I\ O\ O$ we can define a semantic function $\mathcal{R}[\![t]\!](s)$. This semantic function gives us the set of results associated with t, given s. Because the $\texttt{Table}\ I\ O\ O$ data type is inductive, the function $\mathcal{R}[\![\cdot]\!]$ can be defined using structural recursion, as follows:

Definition 17 (Semantics of `Table`)**.**

$$\mathcal{R}[\![\cdot]\!] : \texttt{Table}\ I\ O\ O \to (I \to \mathcal{P}(O)) \to \mathcal{P}(O)$$

$$\mathcal{R}[\![\texttt{Success x}]\!](s) = \{x\} \tag{4.2}$$

$$\mathcal{R}[\![\texttt{Fail}]\!](s) = \emptyset \tag{4.3}$$

$$\mathcal{R}[\![\texttt{Or l r}]\!](s) = \mathcal{R}[\![l]\!](s) \cup \mathcal{R}[\![r]\!](s) \tag{4.4}$$

$$\mathcal{R}[\![\texttt{Rec i k}]\!](s) = \bigcup_{x \in s(i)} \mathcal{R}[\![\texttt{k}(x)]\!](s) \tag{4.5}$$

The two base cases are fairly simple: in the deterministic case (4.2) the result is just a singleton set of the result, and in the failure case (4.3) it is the empty set.

There are two inductive cases as well: a binary choice (4.4) and a recursive call (4.5). A binary choice is handled by taking the union of the results of the left and right branches. A recursive call has an argument i and a continuation k. The result is obtained by finding the set of outcomes in the environment s, and then applying the continuation k to every element x in $s(i)$, and taking the union of the result.

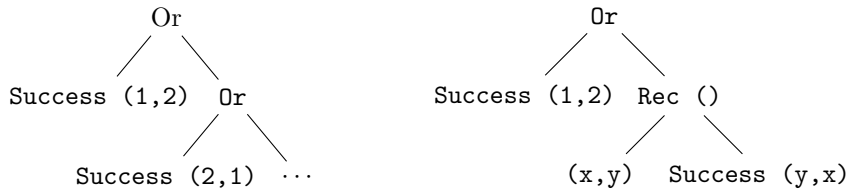
Now we can of define $[\![\cdot]\!]$ as follows: since $\mathcal{R}[\![f(a)]\!]([\![f]\!])$ gives the set of outcomes of $f(a)$ given environment $[\![f]\!]$, and $[\![f]\!](a)$ gives the set of outcomes of $f(a)$, the following must hold about $\mathcal{R}[\![f(a)]\!]([\![f]\!])$:

$$\mathcal{R}[\![f(a)]\!]([\![f]\!]) = [\![f]\!](a) \qquad (\forall a \in I)$$

By η-expanding the left-hand side and using functional extensionality:

$$\lambda a \to \mathcal{R}[\![f(a)]\!]([\![f]\!]) = [\![f]\!]$$

Figure 4.1 Syntax Trees generated by `pairND` and `pairTable ()`.



(a) `pairND`. **(b)** `pairTable ()`.

Hence, $[\![f]\!]$ is a fixed point of $\lambda s \to \lambda a \to \mathcal{R}[\![f(a)]\!](s)$. Now all that remains is to choose a canonical fixed point for $[\![f]\!]$, such that it corresponds to the desired meaning. Note that environments of the type $I \to \mathcal{P}(O)$ (such as $[\![f]\!]$) are ordered by the ordering relation $\sqsubseteq$:

$$f \sqsubseteq g \iff \forall a \in I : f(a) \subseteq g(a)$$

The desired fixed point is *the least fixed point*[6] denoted by $lfp(\cdot)$, when fixed points are ordered by $\sqsubseteq$:

$$[\![f]\!] = lfp(\lambda s \to \lambda a \to \mathcal{R}[\![f(a)]\!](s)) \tag{4.6}$$

To make this more concrete, consider the denotational semantics of `pairTable`:

$$\begin{aligned}
& [\![\texttt{pairTable}]\!](\texttt{()}) \\
= \;& lfp(\lambda s \to \lambda \texttt{()} \to \mathcal{R}[\![\texttt{pairTable ()}]\!](s))(\texttt{()}) && \text{(by (4.6))} \\
= \;& \big(\lambda() \to \{(1,2),(2,1)\}\big)\ () \\
= \;& \{(1,2),(2,1)\} && (\beta\text{-reduction})
\end{aligned}$$

To see why $\lambda() \to \{(1,2),(2,1)\}$ is indeed the least fixed point, consider that it is obviously a fixed point, and moreover, that it is contained in every other fixed point, since:

$$\{(1,2),(2,1)\} \subseteq \mathcal{R}[\![\texttt{pairTable ()}]\!](s)$$

for all environments s.

[6]The least fixed point is well defined, as the function is always continuous (if the syntax tree is finite) and therefore monotone, but the domain $I \to \mathcal{P}(O)$ may not possess the finite ascending chain property, preventing us from computing the solution in a finite amount of time.

4.4.2 Effect Handler Implementation

This section provides a Haskell implementation that corresponds to the denotational semantics from the previous section. This implementation comes in the form of an effect handler (Kiselyov et al., 2013; Schrijvers et al., 2019). Like the denotational semantics, the effect handler is split into two parts: a part that delivers the solution for the entire function by computing a least fixed point (similar to $[\![\cdot]\!]$), and the counterpart of $\mathcal{R}[\![\cdot]\!]$, which, given an environment, computes the results for one syntax tree.

The first part is called `runTable` and the second part is called `go`. The effect handler first computes the least fixed point of the `step` function. The `step` function is a function that takes an environment (of type `Map i (Set o)` from `Data.Map`[7]) and obtains a new environment by running `go` for every entry in the environment. The fixed point of this function is again an environment, in which we look up `i0`, the second argument to `runTable`.

```
runTable :: (Ord i, Ord o)
         => (i -> Table i o o) -> i -> Set o
runTable expr i0 = lfp step s0 ! i0 where
  s0 = M.singleton i0 empty
  step m = foldr (\k -> handle k (expr k)) m (M.keys m)
```

The function `handle`, like $\mathcal{R}[\![\cdot]\!]$ proceeds by case analysis on the syntax tree. However, unlike $\mathcal{R}[\![\cdot]\!]$, `handle` updates an environment instead of just returning a set of solutions. This difference is mainly for programming convenience.

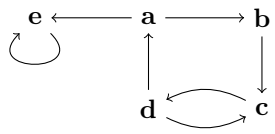
```
handle  ::  (Ord i, Ord o)
    =>  i -> Table i o o
    ->  M.Map i (Set o) -> M.Map i (Set o)
handle i (Success a)  m = M.insertWith union i (singleton a) m
handle i Fail         m = m
handle i (Or l r)     m = handle i r (handle i l m)
handle i (Rec j k)    m
  | Just s <- M.lookup j m = foldr (handle i . k) m (toList s)
  | otherwise              = M.insert j empty m
```

If the syntax tree contains a deterministic result (line 4), we simply add this result to the environment. If the syntax contains a failure (line 5), the environment

[7] Functions and types residing in `Data.Map` are imported with the qualifier `M`, except the lookup operator "`!`".

Figure 4.2 The graph used in the reachable and shortest path example.

```
[a, b, c, d, e] = [ Node "a" [b,e],
                    Node "b" [c],
                    Node "c" [d],
                    Node "d" [c,a],
                    Node "e" [e]]
```



remains unchanged. When presented with a binary choice (line 6), we first update the environment according to the left branch and then according to the right branch. Finally, in the case of a recursive call `Rec j k`, we first check if the map contains an entry for the argument `j` (line 7). If it has no such entry, we add an entry containing the empty set (line 8), otherwise we update the environment based on the existing elements (line 9). Inserting an empty entry for `j`, ensures that the next iteration of `step` also updates the map for `expr j`.

We still need a function `lfp` to iteratively compute a least fixed point. Given an initial value, this function keeps calling itself until the result no longer changes.

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f a0 = let a1 = f a0 in if a1 == a0 then a1 else lfp f a1
```

As an example, consider the semantics of `pairTable`:

```
> runTable pairTable ()
fromList [(1,2),(2,1)]
```

The denotational semantics and its Haskell implementation coincide as expected, but only up to termination (as we show in Section 4.6).

Example 18 (Graph Reachability)**.** We compute reachability in a cyclic directed graph using the non-determinism framework. The example uses an adjacency list representation of a graph where every `Node` has a label (of type `String`) and a list of adjacent nodes:

```
data Node = Node { label :: String, adj :: [Node] }
instance Eq  Node where n1 == n2 = label n1 == label n2
instance Ord Node where n1 <= n2 = label n1 <= label n2
```

Our example graph (see Figure 4.2) consists of five nodes. It contains two cycles: from `c` to `d` and back, and from `e` to itself. Reachability is straightforwardly computed as:

```
reach :: Node -> Table Node Node Node
reach n = return n `Or` (choose (adj n) >>= rec)
```

i.e. `n` is reachable from `n` and all nodes reachable from a neighbor of `n` are reachable from `n`.

```
> runTable reach nodeA
fromList [Node a, Node b, Node c, Node d, Node e]
```

4.5 Dependency Tracking

The effect handler that was provided in Section 4.4 computes the least fixed point in a very naive way, resulting in sub par performance. To illustrate the problem, consider the following program to compute the Fibonacci numbers:

```
fib :: Int -> Table Int Int Int
fib n | n == 0    = return 0
      | n == 1    = return 1
      | otherwise = do f1 <- rec (n - 1)
                       f2 <- rec (n - 2)
                       return (f1 + f2)
```

The evolution of the environment for `runTable fib 2` after each application of `step` is shown in Table 4.1 (ignore column labelled "Cont." for now) Each row in the table corresponds to an iteration of `lfp`. The effect handler recomputes each value in every step: the work performed by an iteration is also performed in all later iterations. In the case of `fib` this leads to an $\mathcal{O}(n^2)$ runtime.[8]

This duplication of work has two sources:

1. The naive least fixed point computation `lfp` iteratively applies the `step` function which folds over *all* keys in the environment, even when the set of outcomes of that key has not changed.
2. The function `go` computes too much: the only part of the syntax tree that is influenced by a recursive call `Rec j k` is the *continuation* `k` of that recursive call. Therefore only the continuation needs to be recomputed.

Although efficiency is not the main focus of this chapter, it would be remiss not to address such a glaring problem. Especially since the solution is simple:

[8] Modulo a logarithmic factor for the use of finite maps.

fib 2		fib 1		fib 0	
Results	Cont.	Results	Cont.	Results	Cont.
∅	mempty				
∅	mempty	∅	(go 2 . k0)		
∅	mempty	{1}	(go 2 . k0)		
∅	mempty	{1}	(go 2 . k0)	∅	(go 1 . l1)
∅	mempty	{1}	(go 2 . k0)	{0}	(go 1 . l1)
{2}	mempty	{1}	(go 2 . k0)	{0}	(go 1 . l1)
{2}	mempty	{1}	(go 2 . k0)	{0}	(go 1 . l1)

Table 4.1: The trace of the environment for `fib 2`.

Figure 4.3 Effect Handler for dependency tracking.

```
1  runTableD :: (Ord i, Ord o) => (i -> Table i o o) -> i -> Set o
2  runTableD expr i0 = go i0 (expr i0) emptyEnv !!! i0 where
3    go i Fail        t = t
4    go i (Or l r)    t = go i r (go i l m)
5    go i (Success x) t | member x (t !!! i) = t
6                       | otherwise          = cont i t x (store i sx t)
7                       where sx = singleton x
8    go i (Rec j k)   t | Just s <- has j t = foldr (go i . k) t' s
9                       | otherwise         = go j (expr j) t'
10                      where t' = register j (go i . k) t
```

both problems can be solved by keeping track of which continuation depends on which recursive call (identified by its argument), and only evaluating the continuation *once* for every new value of the recursive call.

Figure 4.3 shows the code for the interpreter `runTableD` that tracks dependencies. This effect handler relies on seven auxiliary functions defined for the type `Env i o`. This type abstracts over environments that contain continuations in addition to a set of values. Figure 4.4 shows their signatures and implementations.

The computation starts with the call to `go` on line 2, which returns the environment after the least fixed point computed. From this environment the result is extracted using `!!!`. The `go` function proceeds by case analysis: If the syntax tree is a failed computation the environment is unchanged (line 3). If the syntax tree is a binary choice (line 4) the environment is updated first for the left branch, then for the right branch. If the syntax tree is `Success x` and the value `x` is not yet in the environment, the environment is updated by

Figure 4.4 Environment type for dependency tracking.

```
type    Env i o = M.Map i (Set o, C i o)
newtype C   i o = C { runC :: o -> Env i o -> Env i o }

instance Semigroup (C i o) where
  k1 <> k2 = C (\x env -> runC k1 x env <> runC k2 x env)

instance Monoid (C i o) where
  mempty  = C (\x env -> env)
  mappend = (<>)

emptyEnv :: Env i o
emptyEnv = M.empty

store :: (Ord i, Ord o) => i -> o -> Env i o -> Env i o
store i o = M.insertWith mappend i (singleton o, mempty)

has :: Ord i => i -> Env i o -> Maybe (Set o)
has i = fmap fst . M.lookup i

(!!!) :: Ord i => Env i o -> i -> Set o
m !!! i = M.findWithDefault empty i m

cont :: Ord i => i -> Env i o -> (o -> Env i o -> Env i o)
cont i = runC . snd M.findWithDefault mempty i

register :: (Ord i, Ord o) => i -> (o -> Env i o -> Env i o)
         -> Env i o -> Env i o
register i k = M.insertWith mappend i (empty, C k)
```

sequentially applying the continuation that is stored in the environment (line 5-6). If the syntax tree is a recursive call `Rec j k` (line 8-10), and there is an entry for `j` in the environment (line 8) the continuation `k` is registered and `k` is applied to every known result of `j`. Otherwise (line 9), i.e. this is the first time we see a call to `j`, a continuation for `j` is added (line 10) and the recursive function is evaluated for `j`. Otherwise (line 10), a new continuation is added for `j`, and this continuation is applied to every known result of `j`. The least fixed point is reached when `Success` no longer produces new elements.

The trace for `runTableD fib 2` is identical to the trace for `runTable fib 2`, except that behind the scenes only a single entry is computed in each iteration. Moreover, the result of the call to `fib 1` only triggers the continuation `k0` where

```
k0 f1 = \f1 -> Rec 0 (\f2 -> return (f1 + f2))
```

and, the result of `0` only triggers computation of `l1` where

```
l1 f2 = return (1 + f2),
```

instead of the entire syntax tree `fib 2`. To summarise, we obtain a linear runtime for `fib`.

4.6 Lattices

Reconsider the graph defined in Section 4.4. Instead of just finding all reachable nodes, we may want to know the length of the *shortest* path between two nodes. The program `sp` is a first attempt at computing the shortest path between `src` and `dst`:

```
sp :: Node -> Node -> Table Node Int Int
sp dst src | dst == src = return 0
           | otherwise  = do d <- choose (adj src) >>= rec
                             return (d + 1)
```

The denotational semantics gives us the following value for the expression `sp n1 rec n2`:

$$[\![\texttt{sp a rec}]\!](\texttt{b}) = \{2, 4, 6, 8, \ldots\}$$

Since this set is infinite, the effect handler does not terminate. In order to implement a terminating effect handler, we need to first define an alternative denotational semantics that at least produces a finite value, which we can then compute in a finite amount of time.

Lattice Semantics The powersets $\mathcal{P}(O)$ of the previous semantics are generalised to arbitrary complete lattices L. The signature of the new denotational semantics is:

$$[\![\cdot]\!]_L : (I \to \texttt{Table}\ I\ L\ L) \to (I \to L)$$

where $\langle L, \sqsubseteq \rangle$ is a complete lattice.

As before, we first consider the semantics for the case where we already possess an environment $s : I \to L$ containing a (partial) solution for every call $f(a)$, i.e.

$$s(a) \sqsubseteq [\![f]\!]_L(a) \text{ for every } a \in I$$

Then for every syntax tree t : `Table` I L L we can define the denotational semantics $\mathcal{R}[\![\cdot]\!]_L$ in the new setting:

Definition 19 (Lattice Semantics of `Table`)**.**

$$\mathcal{R}[\![\cdot]\!]_L : \texttt{Table}\ I\ L\ L \to (I \to L) \to L \tag{4.7}$$

$$\mathcal{R}[\![\texttt{Success x}]\!]_L(s) = x \tag{4.8}$$

$$\mathcal{R}[\![\texttt{Fail}]\!]_L(s) = \bot \tag{4.9}$$

$$\mathcal{R}[\![\texttt{Or l r}]\!]_L(s) = \mathcal{R}[\![l]\!]_L(s) \bigvee \mathcal{R}[\![r]\!]_L(s) \tag{4.10}$$

$$\mathcal{R}[\![\texttt{Rec j k}]\!]_L(s) = \mathcal{R}[\![\texttt{k}(s(j))]\!]_L(s) \tag{4.11}$$

This semantics is very similar to the previously defined semantics for sets. In particular, the semantics $\mathcal{R}[\![\cdot]\!]$ is almost identical to the semantics $\mathcal{R}[\![\cdot]\!]_L$ specialized to the powerset of the output type ($\mathcal{R}[\![\cdot]\!]_{\mathcal{P}(O)}$). I will make this relationship more precise in Section 4.6.1.

For $[\![f]\!]_L$ we would again like to compute the least fixed point of $\lambda s.\lambda a.\big(\mathcal{R}[\![f(a)]\!]_L(s)\big)$. However, unlike for sets, this function is not guaranteed to be monotonic for arbitrary lattices in general, so a least fixed point might not always exist. Instead, we define the operator $\uparrow$:

Definition 20 (Monotone Iteration)**.** The operator $\uparrow$ is defined as:

$$f \uparrow 0 = \bot$$

$$f \uparrow (n+1) = (f \uparrow n) \bigvee f(f \uparrow n)$$

$$f \uparrow \infty = \bigvee_{n=0}^{\infty} (f \uparrow n)$$

Observe that the sequence $f \uparrow 0$, $f \uparrow 1$, $f \uparrow 2, \ldots$, $f \uparrow \infty$ is non-decreasing.[9]

If the domain of f has the *finite ascending chain property* this sequence is finite. Thus, define $[\![\cdot]\!]_L$ as:

$$[\![f]\!]_L = (\lambda s \to \lambda a \to \mathcal{R}[\![f(a)]\!]_L(s)) \uparrow \infty \tag{4.12}$$

[9]For environments, assume the order $\preccurlyeq$ defined as:

$$s_1 \preccurlyeq s_2 \iff \forall a \in I : s_1(a) \sqsubseteq s_2(a)$$

Moreover, if $\lambda s \to \lambda a \to \mathcal{R}[\![f(a)]\!]_L(s)$ is montone, then $\uparrow$ computes the least fixed point, i.e.

$$[\![f]\!]_L = (\lambda s \to \lambda a \to \mathcal{R}[\![f(a)]\!]_L(s)) \uparrow \infty = \mathit{lfp}(\lambda s \to \lambda a \to \mathcal{R}[\![f(a)]\!]_L(s)) \tag{4.13}$$

(This claim is substantiated in Section 4.6.2.) Otherwise, $\uparrow$ may compute a fixed point that is strictly greater than the least fixed point, or not a fixed point at all.

The semantics defined in this section bears a strong resemblance to the greedy semantics for answer subsumption, defined in Section 3.3.2. Hence, the same caveats apply: the computed result may be unsound (see Example 23).

Example 21 (Shortest Path)**.** The relevant partial order for the shortest path problem is $\langle \mathbb{N} \cup \{\infty\}, \sqsubseteq \rangle$ where

$$a \sqsubseteq b \iff a = \infty \text{ or } b \leq a$$

which is the dual of the canonical order on $\mathbb{N}$, i.e. smaller (shorter) is better. Since the canonical order is well-founded (i.e. has no infinite descending chains), $\sqsubseteq$ has the finite ascending chain property. The order also forms a complete lattice, where a subset X has the following least upper bound:

$$\bigvee X = \begin{cases} \infty & \text{if } X = \emptyset \\ \min_{\sqsubseteq} X & \text{otherwise} \end{cases}$$

In this lattice the least fixed point exists, since the continuation `return (x + 1)` in `sp a` is montone. Simple calculation then gives:

$$[\![\texttt{sp a}]\!]_{\mathbb{N}\cup\{\infty\}}(\texttt{b}) = 3$$

The length of the shortest path from node 2 to node 1 is indeed 3. When a path does not exist, we get the following length:

$$[\![\texttt{sp a}]\!]_{\mathbb{N}\cup\{\infty\}}(\texttt{e}) = \infty$$

Notice the similarity to solving a problem with *Dynamic Programming*. The idea of dynamic programming is to compute the optimal solution to a problem by combining optimal solutions to smaller instances of the same problem. In this setting, recursive calls correspond to instances of the same problem. The lattice allows us to keep only the optimal solution to a given instance, from which the result of a larger instance is computed.

Example 22 (Subset Sum)**.** In this example we solve the following problem: given a (finite) list of integers and an integer s, find the shortest non-empty sublist that sums to s.

Figure 4.5 Code for `Shortest`.

```
data Shortest = None | List [Int] deriving Eq

cons :: Int -> Shortest -> Shortest
cons n Longest   = Longest
cons n (List ns) = List (n:ns)

instance Ord Shortest where
  Shortest <= _ = True
  _ <= Shortest = False
  List xs <= List ys =
    length xs > length ys || (length xs == length ys && xs <= ys)
```

To solve this problem, we need to come up with a (complete) lattice for finite sublists, ordered by *descending* length:

$$\texttt{xs} \leq_{Descending} \texttt{ys} \iff \texttt{length xs} \geq \texttt{length ys}$$

However, this is a pre-order, not a partial order, because lists of the same length may contain different elements or the same elements in a different sequence.

For the purpose of the subset sum problem, these differences do not matter, and there is no clear preference between different lists of the same length that sum to the same value. This apparent impasse between same-length lists can be broken in several ways, for instance, by always choosing the first or last candidate, by enforcing a lexicographical ordering, or by considering equivalence classes of same-length lists. The former is more efficient in practice, while the latter two make more sense order-theoretically. For simplicity, we use a lexicographical order:

$$\texttt{xs} \sqsubseteq_{Shortest} \texttt{ys} \iff \texttt{xs} <_{Descending} \texttt{ys}$$
$$\text{or } \texttt{length xs} = \texttt{length ys} \text{ and } \texttt{xs} \leq_{lex} \texttt{ys}$$

where $\leq_{lex}$ is the lexicographical ordering. This forms a finite semi-lattice, and hence a complete lattice, denoted as $\langle \texttt{Shortest}, \sqsubseteq_{Shortest} \rangle$.

The `Shortest`-datatype in Figure 4.5 also has a special bottom element `None` to indicate the absence of a solution. This figure also defines two functions: `cons` which adds an element to the head of a `List`.

The function `sss` returns the `Shortest` list which sums to its first argument and is a sublist of of its second argument.

```
sss :: (Int,[Int]) -> Table (Int,[Int]) Shortest Shortest
sss (0,[])   = return (List [])
sss (n,[])   = Fail
sss (n,x:xs) = rec (n,xs) `Or` fmap (cons x) (rec (n - x, xs))
```

When the input list is empty we can only construct an empty list with sum 0. When the input list is non-empty, the shortest list summing to `n` is obtained by either recursively searching for a list summing to `n`, or `cons`-ing `x` onto a list that sums to `n - x`.

The function $\mathcal{R}[\![\texttt{sss (n,xs)}]\!]_{\texttt{Shortest}}$ is montone for all (n, xs). Moreover, the environment possesses the finite ascending chain property, and every recursive call decreases the size of the input list by one. Then, given a finite input list, there can only be finitely many recursive calls.

For instance, consider the application of `sss` to `(10,[5,0,5])`:

$$[\![\texttt{sss}]\!](10, [5, 0, 5]) = \texttt{List [5,5]}$$

Indeed `sum [5,5] == 10`. Note that the list `[5,0,5]` itself also sums to 10. However, only the shortest list is retained.

Example 23 (Greedy)**.** Recall the running example from Chapter 3:

```
:- table p(max).
p(0). p(1).
p(2) :- p(1).
p(3) :- p(0).
```

```
?- p(X).
p(2).
```

Rendered in Haskell, we see that it returns the same (incorrect) answer `2` (the right answer is `3`):

```
p () = choice [
  return 0, return 1,
  do {x <- rec (); guard (x == 1); return 2},
  do {x <- rec (); guard (x == 0); return 3}]
```

$$[\![p]\!]_{\{0,1,2,3\}}(()) = 2$$

Thus, like greedy answer subsumption in Prolog, lattice-based tabling behaves unsoundly on certain programs.

Example 24 (Grammar Analysis)**.** Let us demonstrate the added power of the lattice-based semantics by solving a simple grammar analysis problem. Informally, a grammar is a list of rules or productions:

```
type Grammar = [Production]
```

A production is of the form

```
data Production = Symbol :--> [Symbol]
```

There is a single `Symbol` to the left of `:-->`, called the head, and zero or more symbols, the body, on the right hand side. Symbols can be either terminals or non-terminals:

```
type Symbol = Char
isTerminal :: Symbol -> Bool
isTerminal s = not (isUpper s)
```

Terminals may not appear in the head of a rule. Terminals and non-terminals are distinguished by being upper and lower-case characters respectively.

Now, consider the following grammar:

```
grammar :: Grammar
grammar = [ 'E' :--> "T Z",   'E' :--> "(E)",
            'Z' :--> "+ T Z", 'Z' :--> "+ (E)",
            'Z' :--> "",
            'T' :--> "a",     'T' :--> "1"]
```

This grammar describes a simple expression language consisting of one identifier (`a`), one literal (1) and a binary operator (`+`).

The goal is to implement a program that analyses grammars for nullability: a symbol is `Nullable` if it derives the empty string. Nullability is characterized by a function from `Symbol` to `Bool`, using the standard ordering $\leq_{\texttt{Bool}}$ on booleans characterised by:

$$x \leq_{\texttt{Bool}} x \text{ and } \texttt{False} \leq \texttt{True}$$

The partially ordered set $\langle \texttt{Bool}, \leq_{\texttt{Bool}} \rangle$ is a complete lattice, where the least element is `False` and the least upper bound is boolean disjunction.

This leads to the following definition of nullable `nullable`:

```
nullability :: Grammar -> Symbol -> Table Symbol Bool Bool
nullability g s
  | isTerminal s = return False
  | otherwise    = do head :--> body <- choose g
                      guard (head == s)
                      nullables <- mapM rec body
                      return (and nullables)
```

A terminal symbol is never nullable. A non-terminal symbol `s` is nullable, if there exists a production with `s` in the head and with a body consisting of nullable symbols. Note that `s` may also occur in the body.

This gives the following `nullability` for `grammar`:

$$[\![\texttt{nullability grammar}]\!]_{\texttt{Bool}}\texttt{'T'} = \texttt{False}$$

$$[\![\texttt{nullability grammar}]\!]_{\texttt{Bool}}\texttt{'Z'} = \texttt{True}$$

$$[\![\texttt{nullability grammar}]\!]_{\texttt{Bool}}\texttt{'E'} = \texttt{False}$$

These examples demonstrate that a semantics based on lattices allows us to tackle problems we could not previously solve with the set-based semantics. Note that we again arrived at this solution by computing a least fixed point—though of a different function. Furthermore, only the semantics of the syntax has changed, the syntax itself is unmodified. Hence, to implement the semantics only the effect handler needs to be reworked.

4.6.1 Generalized Effect Handler

The semantics $[\![\cdot]\!]_L$ for arbitrary lattices is strictly more general than the semantics for sets $[\![\cdot]\!]$. In particular, we shall see that:

$$[\![f]\!] = [\![\phi(f)]\!]_{\mathcal{P}(O)}$$

where $\phi : (I \to \texttt{Table}\ I\ O\ O) \to (I \to \texttt{Table}\ I\ \mathcal{P}(O)\ \mathcal{P}(O))$ is a function transforming the output types of the function f.

For didactic purposes, the `Table` monad conflates non-determinism (choice and failure) and explicit recursion. However, the relationship between the semantics is much easier to see when interpreting these effects independently. For this reason, the `Table` monad is rebuilt with *modular effect handlers* (Schrijvers et al., 2019), which allows for modular composition of effects and their designated handlers. The elements of this approach are introduced in a side-along fashion, when they are needed.

Syntax Recall that `Table` is a particular instance of a free monad. More generally, free monads have the following shape:

```
data Free sig a = Return a | Op (sig (Free sig a))
```

A `Free` monad forms a syntax tree, where the leaves are a `Return a`, and nodes are impure operations (`Op`), shaped by the signature `sig`. When `sig` is a `Functor`, `Free sig` is a monad:

```
instance Functor f => Monad (Free f) where
  return a = Return a
  Return a >>= f = f a
  Op g     >>= f = Op (fmap (>>= f) g)
```

That is, `return` creates a leaf, and `>>=` grows the tree at the leaves.

For example, consider the signature a non-deterministic choice:

```
data NDF a = FailF | OrF a a deriving Functor
```

Semantics The `Free` monad is interpreted by folding over the structure.

```
fold :: Functor sig => (a -> b) -> (sig b -> b) -> Free sig a -> b
fold gen alg (Return a) = gen a
fold gen alg (Op f)     = alg (fmap (fold gen alg) f)
```

The first argument to the fold, the generator, `gen` is applied to values in the leaves, interpreting a value of type `a` into the *carrier type* `b`. The second argument `alg` is a `sig`-algebra which interprets the signature's operations.

For example, we can interpret `NDF` operations as classic non-determinism:

```
handleList :: Free OrF a -> [a]
handleList = fold gen alg where
  gen x           = [x]
  alg FailF       = []
  alg (OrF xs ys) = xs ++ ys
```

The generator `gen :: a -> [a]` simply wraps values into singleton lists, and the algebra `alg :: OrF [a] -> [a]` is the empty list on failure, or concatenates the choices made in the left and right branches of `OrF`.

```
> handleList (choiceF (return 1) (return 2))
[1,2]
```

Modularity The coproduct `sig1 + sig2` of two signature `Functors` `sig1` and `sig2` is a signature that combines the operations of both:

```
data (f + g) a = Inl (f a) | Inr (g a) deriving Functor
data Void a                            deriving Functor
```

The operations of tabling are expressed by two signatures: `NDF`, defined above, and `RecF`:

```
data RecF i o a = RecF i (o -> a) deriving Functor
```

For convenience, the combined free monad (which is isomorphic to `Table`) is labelled with a type synonym `TableF`, and some smart constructors are defined for it:

```
type TableF i o = Free (NDF + RecF i o)

instance Alternative (TableF i o)
  empty :: TableF i o a
  empty = Op (Inl FailF)

  (<|>) :: TableF i o a ->  TableF i o a ->  TableF i o a
  l <|> r = Op (Inl (OrF l r)))

class Rec i o m | m -> i, m -> o
  recF :: i -> m o

instance Rec i o (TableF i o) where
  recF :: i -> TableF i o o
  recF i = Op (Inr (RecF i Return))
```

When two algebras for different signatures have the same carrier, the coproduct functor can *mediate* between them:

```
(\+/) :: (sig1 a -> a) -> (sig2 a -> a) -> (sig1 + sig2) a -> a
(alg1 \+/ alg2) (Inl op) = alg1 op
(alg1 \+/ alg2) (Inr op) = alg2 op
```

Essentially, `\+/` chooses which algebra to run based on whether it received the left or the right part of the coproduct. When the carrier is a free monad, the left algebra usually does the actual work, and the right algebra merely forwards operations to the carrier.

Effect Handler Implementation for Lattices The typeclass `Lattice` is defined as follows:

```
class Lattice l where
  bottom :: l
  join   :: l -> l -> l
```

This typeclass prioritizes the view of the $\bigvee$-operator as a binary operator. The name `join` comes from the concept of a *join*-semilattice—a partially ordered set where any two elements have a least upper bound. This makes sense from a computational point of view: computing $f \uparrow n$ for finite n only requires a binary $\bigvee$.

Now we replace every `FailF` in a value of type `TableF` with `bottom` and every `OrF` with `join`:

```
runND :: (Functor f, Lattice o) => Free (NDF + f) o -> Free f o
runND = fold return (alg \+/ Op) where
  alg FailF = return bottom
  alg (OrF l r) = join <$> l <*> r
```

The effect handler `runND` effectively eliminates the `NDF` effect from the syntax tree. Only the recursive effect `RecF i o` remains. This effect is eliminated in turn by `runRec`:

```
runRec :: (Ord i, Eq o, Lattice o)
       => (i -> Free (RecF i o) o) -> i -> o
runRec expr i0 = lfp step s0 ! i0 where
  s0      = M.singleton i0 bottom
  step m  = foldr (\k -> fold gen alg (expr k) k) m (M.keys m)
  gen x i = M.insertWith join i x
  alg (RecF j k) i m = k (m' ! j) i m' where
    m' = M.insertWith join j bottom m
```

Note that `runRec` does not expect a plain effect, but an effectful function `i -> Free (RecF i o) o`, and also returns a function `i -> o`. Essentially, `runRec` is a memoizing fixpoint operator applied to the recursive equation defined by `expr`.

Reconsider the shortest path program from Section 4.6:

```
data Dist = Dist Int | InfDist deriving (Eq, Ord, Show)

spF :: Node -> Node -> TableF Node Dist Dist
spF dst src | src == dst = return (Dist 0)
            | otherwise  = do n <- foldr (<|>) empty (adj src)
                              Dist d <- recF n
                              return (Dist (d + 1))
```

Figure 4.6 `Lattice` instances.

```
instance Lattice Dist where
  bottom = InfDist
  join   = min

instance Lattice Bool where
  bottom = False
  join   = or
```

```
instance Lattice Shortest where
  bottom = InfList
  join = argmax len where
  argmax f x y | f x <= f y = y
               | otherwise  = x

instance Ord a => Lattice (Set a) where
  bottom = empty
  join   = union
```

Combining `runND` and `runRec` gives an effect handler `runRec . (runND .)` implementing the semantic function $[\![\cdot]\!]_L$. Figure 4.6 defines the `Lattice`-instance for a number of types, including `Dist`. Now we can run Examples 21, 24 and 22:

```
> runRec (runND . spF n1) n2
Dist 2
> runRec (runND . spF n1) n5
InfDist
> runRec (runND . sss) (10,[5,0,5,2,3])
[5,5]
> runRec (runND . nullability grammar) 'T'
False
```

Lifting Continuations The semantic functions $\mathcal{R}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]_L$ only differ in the case for `Rec j k`: $\mathcal{R}[\![\cdot]\!]$ computes the union of the `k` applied to every element of $s(j)$, while $\mathcal{R}[\![\cdot]\!]_L$ directly applies the `k` to $s(j)$. The similarity between $\mathcal{R}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]_{\mathcal{P}(O)}$ suggests that we can obtain the behaviour of the former with the latter if we just change the continuation in `Rec j k` appropriately. More formally, we want to construct a continuation `k'` such that:

$$\mathcal{R}[\![\texttt{Rec j k}]\!](s) = \mathcal{R}[\![\texttt{Rec j k'}]\!]_{\mathcal{P}(O)}(s)$$

This is achieved by the following code, that lifts a computation in the `Free (RecF i o + NDF)`-monad to one in `Free (NDF + Rec i (Set o))`.

```
liftRec :: Ord o
        => Free (RecF i o + NDF)                 o
        -> Free (NDF      + RecF i (Set o)) (Set o)
liftRec = fold (return . singleton) (alg \+/ fwd) where
  fwd = Op . Inl
  alg (RecF i k) = rec i k' where
    k' = fmap unions . traverse k . toList
```

Composing `liftRec`, `runND` and `runRec` yields a complete effect handler for `Free (NDF + RecF i o)`, corresponding to $[\![\cdot]\!]$:

```
runTableF :: (Ord i, Ord o)
          => (i -> Free (NDF + RecF i o) o)
          -> (i -> Set o)
runTableF f = runRec (runND . liftRec . f)
```

Example 25 (Pairs of Integers)**.** Redefining our running example (Note the instance of `Rec` to account for the different order of the signatures.):

```
instance Rec i o (Free (RecF i o + NDF) where
  recF i = Op (Inl (RecF i Return))

pairF :: () -> Free (RecF () (Int,Int) + NDF) (Int, Int)
pairF () = return (1,2) `orF` do (x,y) <- recF ()
                                 return (y,x)
```

Leads to the following familiar result:

```
> runTableF pairF ()
fromList [(1,2),(2,1)]
```

To recapitulate, we have shown how to implement the effect handler for $[\![\cdot]\!]_L$ for `TableF`. On top of this we implement the effect handler for $[\![\cdot]\!]$ for sets, showing that the semantics for lattices subsumes the semantics for sets.

4.6.2 Monotonicity and the Finite Ascending Chain Property

Recall that in Equation (4.13) I claimed that for a montone function on a complete lattice which has the finite ascending chain property, $f \uparrow \infty$ computes the least fixed point of f.

Loosely, this is a consequence of the fact that every monotone function in a domain with finite ascending chains is ω-continuous. Just like the T_P operator defined in Section 2.1, the least fixed point can then be computed by iteration, due to Kleene's fixed point theorem.

First, however, we need the following lemma:

Lemma 26. *Let $\langle L, \leq \rangle$ be a complete lattice and $f : L \to L$ a monotone function, then for all $n \in \mathbb{N}$:*

$$f \uparrow (n+1) = f(f \uparrow n)$$

Proof. By induction on n:

Case $n = 0$

$$\begin{aligned} f \uparrow (0+1) &= f \uparrow 0 \bigvee f(f \uparrow 0) && \text{(by definition)} \\ &= \bot \bigvee f(f \uparrow 0) && \text{(by definition)} \\ &= f(f \uparrow 0) && (\bot \bigvee x = x) \end{aligned}$$

Case $n + 1$

$$\begin{aligned} & f \uparrow n \leq f \uparrow (n+1) && (f \uparrow (n+1) \text{ is an upper bound}) \\ \Rightarrow & f(f \uparrow n) \leq f(f \uparrow (n+1)) && \text{(monotonicity)} \\ \Rightarrow & f \uparrow (n+1) \leq f(f \uparrow (n+1)) && \text{(induction)} \\ \Rightarrow & f(f \uparrow (n+1)) = f \uparrow (n+1) \bigvee f(f \uparrow (n+1)) && \text{(by def. of the l.u.b.)} \\ \Rightarrow & f(f \uparrow (n+1)) = f \uparrow (n+2) && \text{(by definition)} \end{aligned}$$

□

Corollary 27. *Let $\langle L, \leq \rangle$ be a complete lattice and $f : L \to L$ a monotone function, then for all $n \in \mathbb{N}$:*

$$f \uparrow n = f^n(\bot)$$

Proof. By Lemma 26 and straightforward induction on n:

$$f \uparrow 0 = \bot = f^0(\bot)$$

$$f \uparrow (n+1) = f(f \uparrow n) = f(f^n(\bot)) = f^{n+1}(\bot)$$

□

The main technical result is the following:

Theorem 28 (Equivalence of monotonicity and continuity)**.** *Let $\langle L, \leq \rangle$ be a complete lattice. If $\langle L, \leq \rangle$ has the finite ascending chain property, then a function $f : L \to L$ is monotone if and only if it is ω-continuous.*

Proof. We prove the only-if and if directions separately.

Only-If In general, every ω-continuous function on a complete lattice is monotone. Let $x, y \in L$ such that $x \leq y$, then $x \leq y \leq y \leq \cdots$ forms an ascending chain, and its least upper bound is y, i.e., $x \bigvee y = y$. It follows that $f(y) = f(x \bigvee y) = f(x) \bigvee f(y)$ by continuity. Therefore, $f(y)$ is an upper bound of $f(x)$, hence $f(x) \leq f(y)$.

If Let $a_0 \leq a_1 \leq a_2 \leq \cdots \subseteq L$ be an ascending chain. By the finite ascending chain property, there exists and n_0 such that $\forall n \geq n_0 : a_n = a_{n_0}$. Hence, $\bigvee\{a_n \mid n \geq 0\} = a_{n_0}$.

Moreover, by monotonicity, $f(a_0) \leq f(a_1) \leq \cdots \subseteq L$ is also an ascending chain. In particular, for all $n \geq 0$, it follows from $a_n \leq a_{n_0}$ that $f(a_n) \leq f(a_{n_0})$ Hence,

$$\bigvee\{f(a_n) \mid n \geq 0\} = f(a_{n_0}) = f(\bigvee\{a_n \mid n \geq 0\})$$

□

Corollary 29. *Let $\langle L, \leq \rangle$ be a complete lattice, with the finite ascending chain property. If a function $f : L \to L$ is monotone, then $f \uparrow \infty = lfp(f)$.*

Proof. Since f is ω-continuous by Theorem 28, we have that:

$$\begin{aligned} lfp(f) &= \bigvee\{f^n(\bot) \mid n \geq 0\} && \text{(Kleene's theorem)} \\ &= \bigvee\{f \uparrow n \mid n \geq 0\} && \text{(Corollary 27)} \\ &= f \uparrow \infty && \text{(by definition)} \end{aligned}$$

□

4.7 Mutual Recursion

The syntax we have defined so far has a serious limitation: it can only handle a single function at a time. This section lifts this limitation by extending the syntax to support several, potentially mutually recursive functions within the same non-deterministic computation. Moreover, these functions are allowed to have different argument and return types.

Let us demonstrate the added power with another grammar analysis problem. We implement a program that finds the `First` set of a symbol. A symbol X is

in the `First` set of another symbol Y if any of the strings derived from Y start with X. Solving `First` requires also solving `Nullable`.

Recall that nullability is characterized by a function from `Symbol` to `Any`. First is a also characterized by a function, but one from `Symbol` to `Set Symbol` (recall that `Set` is also a lattice). We encode this fact with a Generalized Algebraic Data Type (GADT) (Jones et al., 2006):

```
data Analyse o where
  Nullable :: Symbol -> Analyse Any
  First    :: Symbol -> Analyse (Set Symbol)
```

The arguments to the *data* constructors `Nullable` and `First` indicate the argument type (in both cases `Symbol`). The argument to the *type* constructor `Analyse` indicates the return type (`Any` or `Set Symbol`).

The syntax data type `TableM` is a variation of `Table`:

```
data TableM i a where
  SuccessM :: a -> TableM i a
  FailM    :: TableM i a
  OrM      :: TableM i a -> TableM i a -> TableM i a
  RecM     :: Lattice o
           => i o -> (o -> TableM i a) -> TableM i a
```

The first three constructors are as before, but the recursive call constructor `RecM` expects an existentially qualified parameter `o` (which must be a lattice). Previously, the syntax introduced `o` in the type, and as such fixes `o` to one particular type for the entire syntax tree. Because of the existential parameter, `TableM` allow calls with different return types. The argument to a recursive call is of type `i o`. When `i` is a GADT, such as `Analyse`, the type `i o` will only be inhabited for the desired `o`s (e.g. `Any` and `Set Symbol` for `Analyse`).

For convenience, four additional smart constructors `recM`, `chooseM`, `choiceM` and `guardM` are also defined. Their meaning is analogous to the smart constructors for `Table`. For brevity, their signature and implementation is omitted here. With these smart constructors we can define explicit recursive calls for `nullable` and `first`.

```
nullable :: Symbol -> TableM Analyse Bool
nullable = fmap getAny . recM . Nullable

first :: Symbol -> TableM Analyse Symbol
first s = recM (First s) >>= chooseM . toList
```

Solving `Nullable` and `First` is straightforward:

```
analyze :: Grammar -> Analyse o -> TableM Analyse o
analyze g (Nullable s)
  | isTerminal s = return (Any False)
  | otherwise    = do
      head :--> body <- chooseM g
      guardM (head == s)
      nullables <- mapM nullable body
      return (and nullables)
analyze g (First s)
  | isTerminal s = return (singleton s)
  | otherwise = do
      head :--> body <- chooseM g
      guardM (head == s)
      nullables <- mapM nullable body
      let (nulls,notNulls) = span snd (zip body nullables)
          prefix = nulls ++ take 1 notNulls
      choiceM [first symbol | (symbol,_) <- prefix]
```

In order to determine the `First` set of a symbol `s`, we need to find the longest nullable prefix of the body for every production with `s` in the head. Then we find the `First` sets of every symbol in the prefix and the symbol directly following the prefix.

A word of caution: the semantics assumes that the output of every input includes $\bot$. This is fine as long as $\bot$ has no more consequences than any other output—that is, the program is monotone. Otherwise this may lead to unexpected or undesirable behaviour. For instance, `nullable` always returns `False` before possibly returning `True`, which is fine for `first` because it only makes the prefix shorter, but becomes problematic, for instance, if we try to find the longest non-nullable prefix. (In this case the solution is to switch to a different lattice, where `False` is not `bot`.) It should be stressed that this behaviour is still consistent with the semantics defined in Definition 19. However, $\mathcal{R}[\![\cdot]\!]_L$ is not monotone.

Implementation The table (`EnvM`) now has the following type:

```
type EnvM i = M.Map (Input i) (Output i)

data Input (i :: * -> *) where
  Input :: Lattice o => i o -> Input i
```

```
  data Output (i :: * -> *) where
    Output :: i o -> o -> Output i
```

The data types `Input` and `Output` wrap a particular instantiation of the `Analyse` type constructor in an existential quantification. The implementation of the `Eq` and `Ord` instances is elided for brevity.

The typeclass `Coerce` to type-safely coerces `Output Analyse` back to `Any` or `Set Symbol`, depending on which constructor of `Analyse` is pattern-matched.

```
class Coerce c where
  coerce :: c o -> Output c -> Maybe o

instance Coerce Analyse where
  coerce (Nullable s) (Output (Nullable t) o) = Just o
  coerce (First s)    (Output (First t)    o) = Just o
  coerce a            o                       = Nothing
```

Lifting `join` to `Output`s:

```
  joinO :: (Coerce i, Lattice o)
        => i o -> Output i -> Output i -> Output i
  joinO i o1 o2 =
    maybe o1 (Output i) (join <$> coerce i o1 <*> coerce i o2)
```

At last we have all the pieces to define the effect handler. The function `goM` evaluates an `TableM i o`, given an environment `EnvM i` and `runTableM` computes the least fixed point of `goM`:

```
  goM :: (Coerce i, Lattice o, Ord (Input i))
      => i o -> TableM i o -> EnvM i -> EnvM i
  goM i (SuccessM x)  m =
    M.insertWith (joinO i) (Input i) (Output i x) m
  goM i FailM       m = m
  goM i (OrM l r)   m = goM i r (goM i l m)
  goM i (RecM j k) m
    | Just s <- M.lookup (Input j) m >>= coerce j = goM i (k s) m
    | otherwise = M.insert (Input j) (Output j bottom) m

  runTableM :: (Ord (Input i), Eq (Output i), Coerce i, Lattice a)
            => (forall o . Lattice o => i o -> TableM i o) -> i a -> a
  runTableM expr i0 =
    fromJust (coerce i0 (lfp step s0 ! Input i0)) where
      s0 = M.singleton (Input i0) (Output i0 bottom)
```

```
    step m = foldr goM' m (M.keys m) where
      goM' (Input k) = goM k (expr k)
```

This implementation is not very different from `runTable` (see Section 4.4), except that some wrapping and unwrapping of `Input` and `Output` is required. Note that dependency tracking is orthogonal to the problem of mutual recursion, i.e. dependency tracking is easily added.

Evaluation Using `runTableM` we can now solve `First`, which requires solving `Nullable`, *in the same computation.*

```
analyzeF :: Grammar -> Symbol -> [Symbol]
analyzeF g = toList . runTableM (analyze g) . First
```

```
> map (analyzeF grammar) ['E','Z','T']
["(1a", "+", "1a"]
```

The computed `First` sets of `E`,`Z` and `T` are the results one would expect from the definition: an expression may start with a parenthesis, a literal or an identifier, and similarly for sub-expressions and terms. The first set can be extended to strings of symbols:

```
firstS :: Grammar -> [Symbol] -> Set Symbol
firstS g s =
 let  (nulls,notNulls) = span analyzeN s
      prefix   = nulls ++ take 1 notNulls
      analyzeN = getAny . runTableM (analyze g) . Nullable
 in unions (map (runTableM (analyze g) . First) prefix)
```

So called *left-recursive conflicts* occur when the `First` sets of bodies of two rules with the same head have a non-empty intersection. The following function `hasConflict` uses `firstS` of rule bodies to detect conflicts:

```
hasConflict :: Grammar -> Bool
hasConflict g =
  let  ruleHead (h  :--> _) = h
       ruleBody (_  :--> b) = b
       ruleConflict [r] = False
       ruleConflict rs  = not (null (foldr1 intersection firsts))
         where firsts = map (firstS g . ruleBody) rs
  in or (map ruleConflict $ groupBy ((==) `on` ruleHead) g)
```

Our example grammar is conflict free:

```
> hasConflict grammar
False
```

However, in the following grammar the rules for 'E' do conflict:

```
grammar2 :: Grammar
grammar2 = ['E' :--> "E + T", 'E' :--> "T",
            'T' :--> "a",     'T' :--> "1"]
> hasConflict grammar2
True
```

Note that the implementation of `firstS` and `hasConflict` is not very efficient: in between calls to `runTableM` the results are simply thrown away. There are two ways to resolve this: either have `runTableM` return the environment `EnvM` to directly perform look-ups, or embed `firstS` and `hasConflict` in `TableM` as well. Such an embedding requires extending `Analyse` with new constructors for `FirstS` and `HasConflict`. This is left as an exercise to the reader. In a more generic setting, one could use a sum-of-products (de Vries and Löh, 2014) approach to solve this problem more satisfactorily.

4.8 Benchmarks

To demonstrate the performance impact of dependency tracking, we evaluate several benchmarks with and without dependency-tracking effect handlers and compare their results.

We consider five problems: computing (large) fibonacci numbers, (a variation of) the knapsack problem, the nqueens problem, computing strongly connected components in a directed graph and finding the shortest path in a directed graph.

The benchmarks were build using the Criterion-benchmark harness and compiled using GHC 7.10.2 (with optimization setting `-O2`) on a 64-bit Linux machine with an Intel Core i7-2600 @ 3.6 Ghz and 16 GB of RAM. All code and results are available in the the Bitbucket repository.[10]

Table 4.2 shows the ordinary least squares regression of all samples of the execution time. The R^2 goodness of fit was above 0.99 in all cases. All values are in milliseconds.

[10] `https://bitbucket.org/AlexanderV/thesis`, in the `IFL2015/benchmarks` directory.

	Size	Naive	Dependency Tracking
Fibonacci	800	455.7	13.77
	805	462.8	13.85
	810	469.3	13.97
Knapsack	10	155.9	8.185
	15	226.6	12.32
	20	309.0	16.04
NQueens	7	4.864	1.259
	9	152.9	26.79
	10	914.8	132.0
SCC	30	57.31	8.843
	33	104.9	10.67
	35	97.97	12.67
Shortest Path	8000	1306	676.6
	8500	1362	718.7
	9000	1451	775.1

Table 4.2: The benchmark results in milliseconds.

For *Fibonacci* the first column is the index of the fibonacci number that is computed. For *Knapsack* it specifies the number of items to choose from (the capacity of the knapsack is fixed to 200). For *NQueens*, it indicates the number of queens that need to be placed on the board. For *SCC* it is the number of nodes in the graph (the number of edges is fixed to half the number of possible edges). Finally, for *Shortest Path* it is the number of edges in the graph.

The results show that effect handlers with dependency tracking massively outperform the naive effect handlers.

4.9 Related Work

Least fixed point computations In their seminal work Scott and Strachey (1971) have introduced the idea of denotational mathematical semantics. In particular, they already define the semantics of programming language syntax based on least fixed points.

Jeannin et al. (2013) give a category theoretic treatment of "Non-well-Founded computations". They give many examples, several of which can be implemented in our non-deterministic framework. Others are beyond our iterative least fixed

point solver, and require more sophisticated techniques, for instance, Gaussian elimination.

The use of least fixed points for solving grammar analysis problems has been extensively treated by Jeuring and Swierstra (1995).

It is a well-known result that dynamic programming can be viewed as memoization of recursive equations. The related work section of Swadi et al. gives an overview (Swadi et al., 2006).

An abstract interpreter computes sound over-approximations of its concrete counterparts. Darais et al. (2017) present a flexible approach for deriving abstract interpreters from an extensible definitional interpreter (Reynolds, 1998). To facilitate abstract interpretation of recursive programs, they describe what is essentially a tabling abstract interpreter. A non-tabling interpreter may not realise that it is stuck in a loop, whereas a tabling interpreter always terminates if the state space of the program is finite (the abstract domain is usually chosen such that it is).

Effect Handlers Algebraic effects as a way to write effectful programs are the subject of much contemporary research (Kiselyov et al., 2013; Kiselyov and Ishii, 2015; Wu and Schrijvers, 2015; Schrijvers et al., 2019).

New languages have been created specifically for studying algebraic effect handlers: *Eff* (Bauer and Pretnar, 2015) extends ML with syntax constructs for effect handlers. Multicore OCaml features a similar system for algebraic effects and handlers, but lacks Eff's type system (Dolan et al., 2015). *Frank* (McBride, 2012) has a Haskell-like syntax and also tracks effect in its type system.

The generalised effect handler that was presented in Section 4.6.1 is not fully modular in the sense of Schrijvers et al. (2019). The encoding for modular effects is somewhat cumbersome, and therefore a more streamlined representation makes more sense didactically (although a modular version was also implemented experimentally).

Non-determinism The classical introduction of non-determinism in functional programming based on the list monad is due to Wadler (Wadler, 1995). Hinze (Hinze, 2000) derives a backtracking monad transformer from a declarative specification. This transformer adds backtracking (non-determinism and failure) to any monad.

Kiselyov et al. (2005) improve on Hinze's work by implementing backtracking monad transformers which support fair non-deterministic choice. They present

three different implementations: the classical approach based on *Streams* (i.e. lazy lists), one based on continuation passing style and another one based on control operators for delimited continuations.

Non-determinism also lends itself to the specification of optimisation problems (Bird and Rabe, 2019). Traditionally, equational reasoning to derive optimisation algorithms from their specifications requires a relational setting (Bird and de Moor, 1997), which is too ponderous a tool for simple equational reasoning about optimisation algorithms. Bird and Rabe (2019) propose to reason about *refinement* of a non-deterministic specification instead: an algorithm should capture a subset of the solutions permitted by the specification. The specification language is a variant of the simply-typed lambda calculus (hence no recursion) extended with a binary choice operator, and equipped with a set-theoretic denotational semantics (similar to Definition 17).

Explicit Recursion I am not the first to tame unbridled recursion by representing recursive calls explicitly. Devriese and Piessens (Devriese and Piessens, 2011, 2012) overcome the limitations of so-called "ω-regular" grammars by representing recursive occurrences of non-terminals in production bodies explicitly.

McBride (McBride, 2015), working in the dependently typed total language Agda,[11] gives a free monadic model for recursive calls. Several possible semantics are discussed, but none based on fixed points.

Oliveira and Cook (2012) extend traditional algebraic datatypes to with explicit definition and manipulation of cycles, offering a practical and convenient way of programming graphs in Haskell. Their approach to explicating recursion differs from this work. Where I use the smart constructor `rec` to indicate a recursive call, they instead use *open recursion*, i.e., functions that take another function as an argument which they call in the recursive position instead of themselves.

An open recursive function can be converted to a `rec`-style one, simply by applying it to `rec`:

```
table :: ((i -> Table i o o) -> i -> Table i o o) -> i -> Set o
table open = runTable (open rec)

pair0 :: () -> Set (Int,Int)
pair0 = table $ \pair0 () -> return (1,2) `Or` do (x,y) <- pair0 ()
                                                  return (y,x)
```

[11] `http://wiki.portal.chalmers.se/agda/pmwiki.php`

While open recursion is a good fit for the explicit graph data structure that Oliveira and Cook (2012) work with, for our purposes (non-deterministic computations) the former style is more convenient, at least for presentation purposes, if not actual programming. When programming, I prefer the open recursive style of combining both the the tabled function and the call to the handler in a single definition (as in `pair0`) but this is, perhaps, in the eye of the beholder.

4.10 Scientific Output

The implementation of tabling in Haskell is available in an online repository in bitbucket (`https://bitbucket.org/AlexanderV/thesis/`).

This chapter draws material from:

> Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. 2015. Fixing non-determinism. In Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (IFL '15). Association for Computing Machinery, New York, NY, USA, Article 5, 1-–12. DOI: `https://doi.org/10.1145/2897336.2897342`

This is a multi-author work, based on the present author's Master thesis. Many of the key definitions are similar, although much of the material was re-worked. Sections 4.6.1 and 4.7 in particular are extensions that were not present in the original Master's thesis. The development in Section 4.6.2 is entirely new to this thesis.

Part II

Probabilistic Programming for Modelling Networks

Chapter 5

Probabilistic Network Modelling

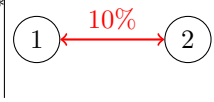
Probabilistic programming languages simplify the creation of probabilistic models. They separate the model from the algorithm that infers probabilities for it (e.g., Church (Goodman et al., 2012), Anglican (Wood et al., 2014), Gen (Cusumano-Towner et al., 2019), ProbLog (Fierens et al., 2015)). Instead of writing a custom procedure tailored to a particular model, the same generic algorithm is used for all programs written in the programming language. Thus, the algorithm can be re-used for many programs, lessening the implementation effort and maintenance burden of the model.

In this work I develop a probabilistic programming language, called P$\lambda\omega$NK.[1] P$\lambda\omega$NK combines diverse features such as higher-order functions, probabilistic choice and parallelism. The main purpose of P$\lambda\omega$NK is to model computer networks and network protocols at an abstract level, and verify a wide variety of properties of such models, e.g., latency, fault-tolerance, or the absence of routing loops. The motivation is the same as for formal verification of computer programs or the mechanical checking of proofs. Namely, during the design of complex networks or protocols, even the best designers are bound to make some mistakes or errors. A computer-checked specification can detect such deficiencies before they end up in a deployed product (Anderson et al., 2014; Foster et al., 2016).

P$\lambda\omega$NK extends Probabilistic NetKAT (PNK) (Foster et al., 2016). Here is a small PNK program:

[1] Pronounced as "plonk".

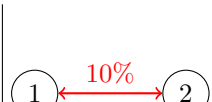
$$\underbrace{(sw = 1;\ \mathbf{drop} \oplus_{0.1} sw \leftarrow 2}_{\text{link 1 to 2}} \,\&\, \underbrace{sw = 2;\ \mathbf{drop} \oplus_{0.1} sw \leftarrow 1)^*}_{\text{link 2 to 1}}$$



The program on the left models the network on the right. This network consists of only two nodes, 1 and 2, with a bidirectional link between them. The link between the nodes is unreliable, e.g., it is a radio link with poor reception. This causes a 10% of the packets to be lost in transit.

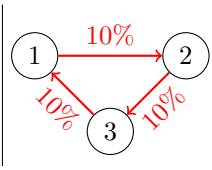
At this point it is not important to understand the meaning of this program exactly. Instead, note that this example already features a lot of repetition: the sub-programs to the left and to the right of the &-operator—modelling the behaviour of the link from 1 to 2 and 2 to 1, respectively—essentially mirror each other. Unfortunately, PNK offers no facilities to take advantage of this insight. The key advantage of P$\lambda\omega$NK over PNK is that we can exploit it by abstracting over the behaviour of both parts with a function:

$$link = \lambda src.\lambda dst.(sw = src; \mathbf{drop} \oplus_{0.1} sw \leftarrow dst)$$
$$(\underbrace{link\ 1\ 2}_{\text{link 1 to 2}} \,\&\, \underbrace{link\ 2\ 1}_{\text{link 2 to 1}})^*$$



The function *link* captures the general forwarding behaviour of the links, independently of a particular source and destination. While this change arguably does not make the program shorter, the improved readability and maintainability make writing and extending the program much more convenient. For instance, adding a third node is now much easier:

$$link = \lambda src.\lambda dst.(sw = src; \mathbf{drop} \oplus_{0.1} sw \leftarrow dst)$$
$$(\underbrace{link\ 1\ 2}_{\text{link 1 to 2}} \,\&\, \underbrace{link\ 2\ 3}_{\text{link 2 to 3}} \,\&\, \underbrace{link\ 3\ 1}_{\text{link 3 to 1}})^*$$



Explicit Syntax for Sequencing Three distinct side-effects are in evidence in the above examples: (1) state—as I explain later, $sw \leftarrow 1$ modifies packets; (2) parallelism—the subprograms for node 1, 2 and 3 are run in parallel with &; and (3) probability—through the $\oplus_{0.1}$-operator. The previous example was carefully chosen such that no arguments to the function contains any side-effects.

Indeed, all arguments are constants, either 1, 2 or 3. Although this is already quite useful, the full language should be more flexible, and also allow applying functions to non-constant expressions. In this case, should the side-effects of this expression be executed before the application and only the resulting value passed to the function (*Call-By-Value*)? Or, should the expression remain unevaluated, allowing the function to decide when to evaluate it (*Call-By-Name*)? Both strategies have their merits and there is no clear winner.

Rather than fix any particular order, I choose to explicitly segregate expressions into *computations* (which have side-effects) and *values* (which do not), loosely inspired by Fine-Grained-Call-By-Value (Levy, 2001). This paradigm partitions expressions into *computations* (which have side-effects) and *values* (which do not). The sequencing of side-effects then becomes explicit: either the computation is evaluated, producing a value which is then passed to the function, or the computation is explicitly turned into a value, by wrapping it in a function.

Semantics P$\lambda\omega$NK has higher-order functions, iteration and probabilistic choice. This significantly complicates the formalisation of the semantics of the language. After all, the standard approach to define denotational semantics for a continuous probabilistic language is based on measure theory. Yet, measure theory does not support general higher-order functions (Aumann et al., 1961), a central feature of the language. For this reason, the denotational semantics of P$\lambda\omega$NK uses (ω-)Quasi-Borel Spaces (Heunen et al., 2017; Vákár et al., 2019) as its domain instead. This is a recently developed alternative axiomatisation of probability theory, which does admit higher-order functions. Moreover, it possesses the ωCPO structure required to model PNK style iteration (Kleene-star).

Another challenge we face is the interaction between higher-order functions, state, parallelism and probability. In P$\lambda\omega$NK, a computation can produce a function in a manner that is simultaneously probabilistic and non-deterministic (parallel), and also locally[2] modifying state. Moreover, we must avoid accidentally duplicating work of parallel branches, since parallel composition (&) is not idempotent (i.e., a program p is *not* equivalent to $p \,\&\, p$). As a result, defining the correct semantics for function application and sequencing is complicated and highly non-trivial. Note that this challenge is present in any calculus that supports function application, probability, parallelism and state—in fact, it is somewhat easier in the present setting, as side-effects cannot occur inside the arguments of a function application.

[2]By local, I mean that the state is not shared between different parallel or probabilistic branches.

As was mentioned earlier, P$\lambda\omega$NK's primary purpose is the *specification* of network models and the verification of properties of those models. It shares this purpose with PNK, which has several computational properties that make it well suited for this purpose: The denotational semantics of PNK can be approximated (Smolka et al., 2017b) computationally, through an iterative procedure. Moreover, at the cost of disallowing the **dup** operation, PNK has decidable program equivalence (Smolka et al., 2017a, 2019). Thus, if we show that a small—hence, easy to prove correct—program is correct, we implicitly show that all equivalent larger programs are also correct.

P$\lambda\omega$NK also possesses these properties, subject to some (minor) restrictions: the approximation procedure exists, if we forbid parallel choice between functions. Essentially, this restricts P$\lambda\omega$NK's parallelism to the parallelism that is present in PNK. Also, I conjecture that disallowing the **dup** operation, as for PNK, results in decidable program equivalence for P$\lambda\omega$NK.

Concretely, this part of the thesis contains the following contributions:

- P$\lambda\omega$NK, a probabilistic programming language for modelling computer networks and protocols (Chapter 6). It features higher-order functions, probabilistic choice and parallelism. P$\lambda\omega$NK extends the earlier probabilistic programming language PNK.
- Unlike PNK, P$\lambda\omega$NK has a (simple) type system (Chapter 6). The type system is important, not only for rejecting invalid programs, but also to ensure that all programs *Strongly Normalise* (Pierce, 2002). On the one hand, strong normalisation indirectly makes the denotational semantics well-defined. On the other hand, this property is handily exploited for compiling P$\lambda\omega$NK to PNK. Recall that P$\lambda\omega$NK is a specification language, and the flexibility of general recursion and real arithmetic is not required.
- P$\lambda\omega$NK is equipped with a denotational semantics (Chapter 8). As P$\lambda\omega$NK contains higher-order functions, iteration and probabilistic choice, the semantics relies on recent advances in the domain theory for probabilistic programs by Vákár et al. (2019). They define an alternative formalisation of probability theory that admits higher-order functions, and iterations, the *ω-Quasi-Borel Spaces* (ωQBSes).

  The well-definedness of this denotational semantics is proven in several theorems. The (pen-and-paper) proofs for several of these theorems use logical relations whose definitions are interesting in their own right. I also prove that this semantics is a conservative extension of PNK's semantics as given by Smolka et al. (2017b). The proofs can be found in the appendix.

- A subfragment of P$\lambda\omega$NK programs which can be compiled into PNK (Chapter 9). Through the type system, the parallelism in P$\lambda\omega$NK is restricted to the parallelism present in PNK. This makes the values that are produced in parallel more predictable, allowing compilation to succeed. Moreover, PNK itself lies entirely within this class. I have mechanised the meta-theory of P$\lambda\omega$NK and the compilation procedure with the aid of the Abella proof-assistant (Gacek, 2008). Theorems bearing a check mark (✓) have been mechanised. The proof scripts are available in the supplementary material (`https://bitbucket.org/AlexanderV/probnetkat-lambda/src/master/abella/`).

  Compilation preserves the denotational semantics of P$\lambda\omega$NK. Since the semantics is a conservative extension, the compiled PNK program behaves identically to the original P$\lambda\omega$NK source program. These theorems are not easily encoded in Abella, and thus have been proven the classical way, with pen and paper.

I have developed a prototype of P$\lambda\omega$NK in Haskell (`https://bitbucket.org/AlexanderV/probnetkat-lambda/src/master/`). The prototype implements an extension to P$\lambda\omega$NK that performs type reconstruction. The implementation can be used to run the examples that are presented in this thesis.

The remainder of this part of the thesis is organised as follows: Chapter 6 briefly reviews PNK, presents the key design ideas of P$\lambda\omega$NK and defines P$\lambda\omega$NK's syntax and type system. Chapter 7 provides background material on ωQBS. Chapter 8 equips P$\lambda\omega$NK with a denotational semantics, and discusses its properties. Chapter 9 defines the procedure for compiling P$\lambda\omega$NK programs into PNK, and discusses its meta-theory.

This part of the thesis draws material from:

> Alexander Vandenbroucke and Tom Schrijvers. 2019. P$\lambda\omega$NK: functional probabilistic NetKAT. Proc. ACM Program. Lang. 4, POPL, Article 39 (January 2020), 27 pages. DOI: `https://doi.org/10.1145/3371107`

In this multi-author work, Tom Schrijver's role is mostly limited to collaboration on the high-level design of P$\lambda\omega$NK, and assistance with some of the proofs. The author of this thesis is responsible for the syntax, type system, semantics and compilation procedure, and mechanisation of the meta-theory in Abella.

Chapter 6

Functional Probabilistic NetKAT

This chapter covers the key design ideas of P$\lambda\omega$NK and its syntax and type system. I begin, however, with a brief introduction to Probabilistic NetKAT, on which P$\lambda\omega$NK is based.

6.1 A Brief Introduction to Probabilistic NetKAT

The central notion of PNK are *packet histories*, i.e., ordered sequences of packets. We write $\pi :: h$ for a history consisting of its most recent packet π, followed by the earlier history h. The empty history is written as $\langle\rangle$. The set of all packet histories is denoted PH. PNK programs operate on sets of these histories.

Packets themselves are intended to be a simplified model of real-world binary network packets. As such, they consist of a number of header fields, which are assigned numeric values. Contrary to real-world packets, they do not contain a payload because it is irrelevant for routing decisions. The examples in this thesis commonly use the following headers: the switch the packet is currently at (sw) and the port the packet is currently at (pt).

PNK programs are constructed by composing a number of primitive operations. These primitives are predicates (e.g., **drop** or $sw = 1$), assignments (e.g., $pt \leftarrow 2$), and duplication (**dup**). Recall that PNK programs operate on sets of packets histories. *Predicates* filter this set, allowing only specific histories.

For instance, tests such as $sw = 1$ only allow histories where the first packet's header sw is set to 1, whereas **drop** denies all packets, producing an empty set. *Assignments* instead modify the histories in the set. For instance, the expression $pt \leftarrow 2$ sets the pt header to 2 for the first packet of every history. *Duplication* **dup** duplicates and prepends the first packet of every history in the set, i.e. for history $\pi :: h$, **dup** produces a history $\pi :: \pi :: h$.

Primitive operations can be composed in three ways:[1] sequentially, parallelly or probabilistically. *Sequential composition* $(;)$ executes both operations one after the other, the output of the first becoming the input of the second. For instance, $sw = 1;\ pt \leftarrow 2$ first filters out all histories where the sw header of the first packet is not 1, and then modifies the pt header of the first packet of every history. *Parallel composition* $(\&)$ executes both operations independently and then takes the union of the resulting sets. For instance, $sw = 1 \,\&\, sw = 2$ allows only packet histories that have the first packet's header set to either 1 or 2. Finally, *probabilistic choice* $(\oplus_r)$ chooses either the left side with probability r, or the right with probability $1 - r$. For instance $sw \leftarrow 1 \oplus_{0.5} sw \leftarrow 2$, probabilistically chooses to set the first packet's header to 1 or to 2. If we sample from this expression, we see a set where the first packets' headers are either all set to 1, or all set to 2. The probability of either event is exactly 50%.

6.2 Modelling in PNK

Let us consider how to model the network shown in the top-left corner of Figure 6.1 in PNK. The network consists of four nodes, with links from node 1 to nodes 2 and 3, and from nodes 2 and 3 to node 4. The objective is to send a network packet from node 1 to node 4, routed through either node 2 or node 3.

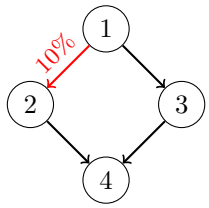
In order to accurately model the network, the program must model two independent aspects: the *network topology*, that is, the links connecting the nodes (and their behaviour), and the *routing programs* running on the nodes, receiving and forwarding incoming packets. The PNK program modelling the network is shown on the left of Figure 6.1.

Topology The topology is captured by the term t on the left-hand side of Figure 6.1. It is a parallel composition of terms. It models, from top to bottom, the links from 1 to 2, from 1 to 3, from 2 to 4 and from 3 to 4. Each of the links consists of a number of statements, composed sequentially. For each link,

[1] Actually, there is a distinction between composition for predicates and other operations, but it is not relevant here.

Figure 6.1 PNK and P$\lambda\omega$NK models of a small network consisting of 4 nodes.

Network



PNK

Topology

$$
\begin{aligned}
t = \big(&sw = 1;\ pt = 2;\\
&(sw \leftarrow 2;\ pt \leftarrow 1) \oplus_{0.9} \mathbf{drop}\big)\\
\&\ &(sw = 1;\ pt = 3;\ sw \leftarrow 3;\ pt \leftarrow 1)\\
\&\ &(sw = 2;\ pt = 4;\ sw \leftarrow 4;\ pt \leftarrow 2)\\
\&\ &(sw = 3;\ pt = 4;\ sw \leftarrow 4;\ pt \leftarrow 3)
\end{aligned}
$$

Routing

$$
\begin{aligned}
p = &(sw = 1;\ pt \leftarrow 2)\\
\&\ &(sw = 2;\ pt \leftarrow 4)
\end{aligned}
$$

P$\lambda\omega$NK

Prelude

$$
\begin{aligned}
send &= \lambda src.\lambda dst.(sw = src;\ pt = dst)\\
recv &= \lambda src.\lambda dst.(sw \leftarrow dst;\ pt \leftarrow src)\\
link &= \lambda src.\lambda dst.\\
&\quad (send\ src\ dst;\ recv\ src\ dst)
\end{aligned}
$$

$$
\begin{aligned}
forward &= \lambda src.\lambda act.\\
&\quad (sw = src;\ act\ \mathbf{unit})\\
to &= \lambda dst.(pt \leftarrow dst)
\end{aligned}
$$

Topology

$$
\begin{aligned}
t = (&send\ 1\ 2;\ (recv\ 1\ 2 \oplus_{0.9} \mathbf{drop}))\\
\&\ &link\ 1\ 3 \,\&\, link\ 2\ 4 \,\&\, link\ 3\ 4
\end{aligned}
$$

Routing

$$
\begin{aligned}
p &= forward\ 1\ (\lambda.(to\ 2))\\
&\ \&\ forward\ 2\ (\lambda.(to\ 4))\\
p' &= forward\ 1\ \underline{(\lambda.(to\ 2 \oplus_{0.5} to\ 3))}\\
&\ \&\ forward\ 2\ (\lambda.(to\ 4))\\
&\ \&\ \underline{forward\ 3\ (\lambda.(to\ 4))}
\end{aligned}
$$

Main Expression

$$(p;\ t)^*;\ sw = 4$$

it is verified that a packet is actually at the origin of the link by a series of guards (e.g., $sw = 1;\ pt = 3$, for the link from 1 to 3). Then, the packet is modified, setting sw and pt to the next switch on the link (e.g. $sw \leftarrow 3;\ pt \leftarrow 1$), thus transmitting the packet across the link. For the link between 1 and 2, we model unreliability by making a probabilistic choice ($\oplus_{0.9}$), choosing normal transmission 90% of the time and dropping (with **drop**) the packet 10% of the time. Packets that do not match the switch and port are rejected before their headers are modified.

Routing The routing program is captured by the term p, a parallel composition of the routing programs for each node. For each node, it is verified that the

sw field matches the node, and a node-specific routing program is then run: at node 1, packets are forwarded to node 2 and at node 2 packets are forwarded to node 4. Node 3 is unused for now.

Main Expression The main expression (at the bottom of Figure 6.1) combines topology and routing, and provides an exit predicate. The $(\cdot)^*$, Kleene star, means iteration. Thus, the program p; t is repeated until the exit predicate $sw = 4$ is satisfied. This predicate checks that the packet has arrived at node 4, its destination.

Having a PNK model of our network allows us to estimate the probability of certain queries, such as the probability that a packet reaches its destination (90% in this case) or measure the expected congestion. This is by no means an exhaustive list. A slightly modified program permits us to estimate the latency (i.e., the average length of a path), or by restricting the language, program equivalence becomes decidable, creating an easy way to verify correctness (Smolka et al., 2017a, 2019). For additional examples and details, the interested reader should consult the earlier work of Foster et al. (2016).

6.3 Extending PNK with functions

Even the small example from the previous section is quite tedious and repetitive to write. The root cause of this issue is the complete lack of abstraction facilities in PNK. It is well-known that abstraction improves modularity and enables code re-use.

Arguably one of the most basic abstraction facilities available in programming languages is the venerable λ-abstraction. In P$\lambda\omega$NK, which additionally supports λ-abstractions, we instead encode the network topology (which features a lot of repetition) more concisely.[2] We can identify several recurring patterns and give them appropriate names. These are shown in the top-right corner of Figure 6.1. The primitive patterns are sending and receiving, which are combined to create a link. Functions such as *send*, *recv* and *link* could be defined in a library or a language prelude, to be reused by other programs.

We can now re-write the topology as show on the right of Figure 6.1. The links from 1 to 3, 2 to 4 and 3 to 4 simply call the appropriate function. The link from 1 to 2 must directly rely on sending and receiving, but even here, we can see the benefits of the approach in reducing duplication.

[2]The code presented here is untyped, for didactic purposes. From Section 6.4 onwards we will use typed P$\lambda\omega$NK, although type reconstruction for P$\lambda\omega$NK is not difficult.

The functional re-write of the routing program p features a more advanced use of functions (program p on the right-hand side of Figure 6.1). The function *forward* takes a source node (*src*) and a forwarding action *act* to perform. This action is a *thunk*, a suspended computation, which can be executed or *forced*, by applying it to a **unit** value. The justification for these constructions is explained in the next section. Since thunks are functions, *forward* is a higher-order function. A more extensive use of higher-order functions can be found in the Gossip Protocols example (Example 54).

The action *act* uses the function function *to*, which sets the packet's *pt* header to the given destination *dst*.

In p, the forwarding program for node 1 calls *to*, immediately suspends the call (using **thunk**), and passes it to *forward*. Forwarding for node 2 proceeds in a similar fashion.

The main expression is as before. Due to the additional structure, the readability of the program has improved considerably. Furthermore, code re-use has gone up, making the program easier to change. For instance, to model an additional link, from 1 to 4, we only need to add the call (*link* 1 4), instead of the more lengthy ($sw = 1$; $pt = 4$; $sw \leftarrow 4$; $pt \leftarrow 1$).

For another example, suppose we want to change the forwarding behaviour of our network, such that node 1 now chooses to forward to either node 2 or node 3 with equal probability, then we need only extend p slightly, obtaining p'. The necessary additions have been highlighted in Figure 6.1. In short, this section demonstrates the advantage for readability and maintainability that P$\lambda\omega$NK provides.

6.3.1 Explicit Thunks and Sequencing

As was shown in the previous section, it is highly desirable that functions are higher-order, in the sense that functions—and PNK expressions—can occur as arguments to other functions (e.g., *forward*). Then it seems reasonable to expect to be able to write the following:

$$sw \leftarrow 0;\ (\lambda x.sw = 1)\ (sw \leftarrow 1)$$

However, this presents an issue, since $sw \leftarrow 1$ has a side-effect: it sets a header in the packet. The evaluation order is now important: if $sw \leftarrow 1$ is evaluated before the application, this program accepts the packet, otherwise it drops the packet. The former corresponds to a Call-By-Value (CBV) order, the latter to a Call-By-Name (CBN) order. There is no clear reason to prefer one over the other, and both are useful in practice.

Indeed, P$\lambda\omega$NK does not fix any particular evaluation order. Instead, it uses the Fine-Grained Call-By-Value (FG-CBV) paradigm (Levy, 2001), which subsumes both. In FG-CBV, terms are segregated into values and computations. Computations can be evaluated (possibly with side-effects), as opposed to values, which cannot be directly evaluated. Functions (classified as values themselves) can only be applied to values. PNK terms are also computations, so the expression above is invalid FG-CBV syntax, since $sw \leftarrow 1$ is not a value.

Instead, we obtain two possible variants, depending on whether CBV or CBN is intended (here the *to* is a primitive, *not* the function *to* defined earlier):

$$\begin{array}{ll} \textbf{CBV:} & sw \leftarrow 0;\ (sw \leftarrow 1)\ \textbf{to}\ y.((\lambda x.sw = 1)\ y) \\ \textbf{CBN:} & sw \leftarrow 0;\ (\lambda x.sw = 1)\ (\lambda x.(sw \leftarrow 1)) \end{array}$$

In the first case, the FG-CBV primitive *to* evaluates $sw \leftarrow 1$, including side effects, and binds the value that is produced to the variable y, followed by applying the function to y. In the second case, $sw \leftarrow 1$ is thunked (wrapped in an abstraction to delay evaluation), and the function is applied to this thunk instead.

6.3.2 Semantics of Iteration

Foster et al. (2016) define the semantics of PNK in terms of measure theory. Semantically, a program denotes a function that maps sets of packet histories to a probability distribution over sets of packet histories. For example, the program $src = 1$, given input set A, returns a probability distribution that has probability 1 at the set $\{\pi :: h \in A \mid \pi.sw = 1\}$ (the notation $\pi.sw$ refers to the value of the sw header of π) and is zero everywhere else.

Iteration (*) is defined as an infinite stochastic process. The formalisation of this process is quite involved. Smolka et al. (2017b) give an equivalent, but much simpler definition, based on standard notions from domain theory.

This work, retains this much simpler second definition, but extends it to support higher-order functions. However, since their domain is measure-theoretic, it does not support such functions, hence their domain is not suitable for P$\lambda\omega$NK.

Instead, ω-Quasi Borel Spaces are adopted as the domain of choice. In contrast to standard measure theory, ω-Quasi Borel Spaces, developed by Vákár et al. (2019), have domain-theoretic structure, support measure-theory-like operations and admit higher-order functions.

6.3.3 Key Ideas

Denotational Semantics P$\lambda\omega$NK's denotational semantics is defined within the ωQBS framework. In addition to PNK's semantics, it also manages variable environments, and passes values (e.g. headers, constants, thunks) instead of just sets of packet histories.

The well-definedness of the semantics depends on two properties: First, to apply the domain-theoretic approach, we need to show that a specific notion of *continuity* holds. Second, we need a property that is similar to, or a consequence of, *strong normalisation*, but for denotational semantics. Informally, this property says that the denotation of a program produces only finitely many distinct values in a parallel fashion. The proofs for both properties have a similar structure: they proceed by induction on typing derivations of P$\lambda\omega$NK programs and make use of logical relations. The definitions of these logical relations are interesting in their own right (see Sections 8.3 and 8.4).

Approximation and Decidable Equivalence The streamlined semantics of Smolka et al. (2017b) formalises an iterative approximation procedure for PNK programs. The idea is to expand the iterations (*) up to n times, for some finite n. Chapter 9 defines a procedure to compile a P$\lambda\omega$NK program to PNK, while preserving the denotational semantics. Because the semantics is conservative with respect to the semantics of PNK, we can also approximate the compiled program. Unfortunately, this compilation is only valid for a subclass of P$\lambda\omega$NK programs. Essentially, the trick is to impose additional restrictions in the type system, such that the parallelism in P$\lambda\omega$NK is limited to the parallelism that occurs in PNK.

Moreover, without the **dup** operation, PNK exhibits decidable program equivalence (Smolka et al., 2017a). The **dup** operation duplicates the packet at the head of a packet history. Thus, removing this operation restrains all packet histories to the same length, making the state space of a program essentially discrete and finite. I conjecture that the same restriction also makes P$\lambda\omega$NK's state space discrete. However, under this restriction, PNK produces only discrete distributions. Many useful properties are expressible in this sub-language (Smolka et al., 2019). However, it cannot express some relevant properties, e.g. latency. Hence, this setting is less interesting than the full language. For this reason, I focus on full P$\lambda\omega$NK. I revisit these issues in Chapter 9.

Figure 6.2 P$\lambda\omega$NK syntax.

Terms

$$
\begin{array}{rcll}
x, y & \in & \mathit{Var} & \text{(Variables)} \\
h & \in & \mathit{Headers} & \text{(Header names)} \\
n & \in & \{1, \ldots, N\} & \text{(Header values)} \\
r & \in & \mathbb{R} & \text{(Weights)} \\
V & = & x \mid \mathbf{unit} \mid h \mid n \mid \lambda x{:}T.C & \text{(Values)} \\
P & = & \mathbf{skip} \mid \mathbf{drop} \mid V = V \mid \neg P \mid P \wedge P \mid P \vee P & \text{(Predicates)} \\
C & = & \langle P \rangle \mid V \leftarrow V \mid \mathbf{dup} \mid C; C \mid C \,\&\, C & \text{(PNK Computations)} \\
 & \mid & C \oplus_r C \mid C^* & \text{(PNK Computations cont.)} \\
 & \mid & \mathbf{produce}\ V \mid C\ \mathbf{to}\ x.C \mid V\ V & \text{(New Computations)}
\end{array}
$$

Types and Contexts

$$
\begin{array}{rcll}
T & = & \mathbf{1} \mid \mathcal{H} \mid \mathbb{N} \mid T \to T & \text{(Types)} \\
\Gamma & = & \emptyset \mid x{:}T, \Gamma & \text{(Contexts)}
\end{array}
$$

6.4 Syntax of Terms

The syntax of P$\lambda\omega$NK (Figure 6.2) is a straightforward extension of the syntax of PNK with higher-order functions and sequencing. The syntax terms are segregated into values and computations.

Values V are either variables x, **unit** values, header names h (from a finite set *Headers*, e.g. *sw*, *pt*, ...), literals n (natural numbers, the values that can be assigned to a header) or functions. Values are never evaluated, but functions can be applied.

Computations C, on the other hand, can be evaluated, but cannot occur as the argument to a function. Consequently, only values can appear on the right-hand side of an application.

All original PNK constructs are computations C. *Predicates* P are a subsort of those. *Atomic* predicates are **skip**, **drop** or tests $V = V$. *Composite* predicates are negation $\neg$, conjunction ($\wedge$) or disjunction ($\vee$).[3] Predicates are explicitly

[3]Contrary to previous work (Foster et al., 2016), I do distinguish the syntax for disjunctive and conjunctive predicates from parallel and sequential composition of computations, for improved clarity. Earlier developments used the same operators for both.

turned into computations with $\langle\cdot\rangle$. The remaining features inherited from PNK are (non-predicate) computations in P$\lambda\omega$NK. They assign values to headers $V \leftarrow V$, duplicate packets with **dup**, compose sequentially $(;)$ or in parallel $(\&)$, make a probabilistic choice $\oplus_r$ (I sometimes elide the weight r when it is not relevant) or iterate *. The computations that P$\lambda\omega$NK inherits from PNK are referred to as *probabilistic* computations.

Finally, P$\lambda\omega$NK adds the following computations to PNK: producing a value V (**produce** V), sequencing computations with C_1 **to** $x.C_2$ and applying functions V_1 to values V_2 with $(V_1\ V_2)$.

Terminal computations are those computations that cannot be evaluated further:

$$\begin{aligned} R \;&=\; P \mid V \leftarrow V \mid \mathbf{dup} \mid R;\, R \mid R \,\&\, R \mid R \oplus_r R \mid R^* \\ &\mid\; \mathbf{produce}\ V \end{aligned}$$

6.5 Types and Type System

Because PNK terms cannot "go wrong" or get stuck, the language did not come with a type system. This is no longer true for P$\lambda\omega$NK, which introduces stuck terms with the lambda calculus. For this reason, P$\lambda\omega$NK is enriched with a simple type system.

The chief reason for choosing *simple* types is strong normalisation (see Theorem 53). In order to not compromise P$\lambda\omega$NK's suitability as a modelling language, certain properties, such as approximation and program equivalence must remain decidable, requiring terminating reduction for all well-typed programs.

For simplicity, I elide other less essential features, such as sum and product types, but such features could be added without too much trouble.

The bottom part of Figure 6.2 shows the syntax of types. There are 4 forms of types: unit types $\mathbf{1}$, header labels $\mathcal{H}$, header literals $\mathbb{N}$ and function types $T \to T$.

Figure 6.3 shows the typing rules. The type system defines two mutually recursive typing judgements: $\Gamma \vdash_v V : T$ for typing value terms, and $\Gamma \vdash_c C : T$ for typing computation terms. The former should be read as "in context Γ, value V *has* type T", whereas the latter is "in context Γ, computation C *produces* a value of type T".

The rules for values are straightforward. Well-formedness for predicates is defined by the relation $\Gamma \models \mathbf{wf}\ P$. Well-formed predicates give rise to

Figure 6.3 P$\lambda\omega$NK's typing rules.

$$\boxed{\Gamma \vdash_v V : T}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_v x : T} \qquad \frac{}{\Gamma \vdash_v \mathbf{unit} : \mathbf{1}} \qquad \frac{}{\Gamma \vdash_v h : \mathcal{H}} \qquad \frac{}{\Gamma \vdash_v n : \mathbb{N}}$$

$$\frac{x : T_1, \Gamma \vdash_v C : T_2}{\Gamma \vdash_v \lambda x : T_1 . C : T_1 \to T_2}$$

$$\boxed{\Gamma \models \mathbf{wf}\ P}$$

$$\frac{}{\Gamma \models \mathbf{wf\ skip}} \qquad \frac{}{\Gamma \models \mathbf{wf\ drop}} \qquad \frac{\Gamma \vdash_v V_1 : \mathcal{H} \qquad \Gamma \vdash_v V_2 : \mathbb{N}}{\Gamma \models \mathbf{wf}\ V_1 = V_2}$$

$$\frac{\Gamma \models \mathbf{wf}\ P}{\Gamma \models \mathbf{wf}\ \neg P} \qquad \frac{\Gamma \models \mathbf{wf}\ P_1 \qquad \Gamma \models \mathbf{wf}\ P_2}{\Gamma \models \mathbf{wf}\ P_1 \wedge P_2} \qquad \frac{\Gamma \models \mathbf{wf}\ P_1 \qquad \Gamma \models \mathbf{wf}\ P_2}{\Gamma \models \mathbf{wf}\ P_1 \vee P_2}$$

$$\boxed{\Gamma \vdash_c C : T}$$

$$\frac{\Gamma \models \mathbf{wf}\ P}{\Gamma \vdash_c \langle P \rangle : \mathbf{1}} \qquad \frac{\Gamma \vdash_v V_1 : \mathcal{H} \qquad \Gamma \vdash_v V_2 : \mathbb{N}}{\Gamma \vdash_c V_1 \leftarrow V_2 : \mathbf{1}} \qquad \frac{}{\Gamma \vdash_c \mathbf{dup} : \mathbf{1}}$$

$$\frac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C_1;\ C_2 : T_2} \qquad \frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 : T}$$

$$\frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \oplus C_2 : T} \qquad \frac{\Gamma \vdash_c C : \mathbf{1}}{\Gamma \vdash_c C^* : \mathbf{1}} \qquad \frac{\Gamma \vdash_v V : T}{\Gamma \vdash_c \mathbf{produce}\ V : T}$$

$$\frac{\Gamma \vdash_c C_1 : T_1 \qquad x : T_1, \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C_1\ \mathbf{to}\ x . C_2 : T_2} \qquad \frac{\Gamma \vdash_v V_1 : T_1 \to T_2 \qquad \Gamma \vdash_v V_2 : T_1}{\Gamma \vdash_c V\ V : T_2}$$

computations of type **1** because they do not produce useful results, only useful side-effects. This also applies to the rules for modification and duplication.

When sequentially composing computations C_1; C_2, only C_2 determines the type of the whole computation. For parallel composition and choice, the types of both computations must be *the same*. Intuitively, the former discards the value produced by the first computation, while the latter two must somehow

combine the two produced values, requiring them to have the same type.

Iteration C^* has the same type as C, which is only allowed to be $\mathbf{1}$, for the following reason. Consider that the iteration could be zero or more times, and thus in the zero case would require inventing a value of an arbitrary type, which we cannot do unless we restrict iteration to a fixed type with a known value—the unit type. The remaining rules in Figure 6.3 are standard for FG-CBV (Levy, 2001).

The previous sections, also contained "top-level" definitions of the form $f = \cdots$. These are to be understood as syntactic sugar. They desugar in two steps as follows:

$$\begin{array}{l} f = C_1 \\ C_2 \end{array} \longrightarrow \begin{array}{l} f = \lambda x{:}\mathbf{1}.C_1 \\ [f \mapsto (f\ \mathbf{unit})]C_2 \end{array} \qquad \begin{array}{l} f = V \\ C \end{array} \longrightarrow \lambda f{:}T.C\ V$$

where x is fresh, and $\vdash_c V{:}T$. The first step desugars all top-level computations into a value, by wrapping it into a function. In the second step, all top-levels are values, and C is turned into a function applied to these values.

6.6 Related Work

Software Defined Networks Software Defined Networks (SDN) (Foster et al., 2013) aim to decrease the complexity of the modern computer networking environment, by offering a clean open interface between heterogeneous networking devices (e.g. routers, switches and firewalls). This is accomplished through the OpenFlow[4] protocol. Unfortunately, this is a rather low-level protocol making it inconvenient to program hardware in OpenFlow directly.

The aim of the Frenetic project (Reich et al., 2013) is to design the right high-level abstractions for controlling OpenFlow hardware. NetKAT (Anderson et al., 2014) and PNK (Foster et al., 2016) were developed under this project. Some preliminary case studies were performed with PNK, modelling the behaviour of several traffic engineering approaches. The effort involved in these case studies was not reported. Interestingly, the few samples of code that were provided seem to use features (e.g. finite iterations, variables), that are not part of the formalised fragment of PNK, but could be implemented fully within P$\lambda\omega$NK.

Another project in this space is the P4 language (Bosshart et al., 2014), supported by the P4 language consortium, which focuses less on verification, but more on efficient compilation for programs to be installed on data-plane hardware.

[4] `https://www.opennetworking.org/`

Probabilistic Programming The goal of probabilistic programming (Goodman, 2013) can be captured by the following equation:

$$\text{PPL} = \text{MODELLING LANGUAGE} + \text{INFERENCE ALGORITHM}$$

That is, probabilistic programming's goal is to unify probabilistic modelling and general purpose programming: probabilistic models are written in the language, and the probabilities are inferred using a generic inference algorithm.

Stan (Carpenter et al., 2017) is a very popular statistical modelling language, with bindings to R, Python, MATLAB, Julia and several others. More recently, languages such as Gen (Cusumano-Towner et al., 2019) and Turing (Ge et al., 2018) have started to make the inference algorithms programmable, in addition to the model, since fine-tuning the inference can lead to large performance gains.

Probabilistic Programming has been applied to such diverse problems as 3D body pose estimation from depth data (Cusumano-Towner et al., 2019), genetics (De Maeyer et al., 2013) and Automatic Video Montage (Aerts et al., 2016).

Functional PPLs such as Anglican (Wood et al., 2014), Venture (Lu, 2016), or Gen (Lu, 2016), allow distributions over higher-order functions. However, unlike this work, they do not formalise the semantics of higher-order functions, focusing instead on language design and implementation, and inference. The work on quasi-Borel Spaces is at least partially motivated by the unfilled need for a theoretical foundation for these languages Heunen et al. (2017). Quasi-Borel Spaces have been used by Scibior et al. (2018) to verify the correctness of modular Bayesian inference algorithms.

Kleene Algebra with Test Probabilistic NetKAT is a probabilistic development of the modelling language NetKAT (Anderson et al., 2014). In turn, KAT stands for Kleene Algebra with Test (Kozen, 1997; Kozen and Smith, 1996), a generalisation of regular expressions. Like regular expressions, KATs have an associated theory of languages and automata. They have been applied amongst others to program verification, concurrency and, of course, networks. Recently, Guarded KATs (GKATs) have been developed by Smolka et al. (2020). In a GKAT equivalence of two expressions is efficiently decidable—unlike general KAT, where it may be undecidable. There is a GKAT version (McNetKAT) of PNK, which features this capability (Smolka et al., 2019).

Chapter 7

A Convenient Category for P$\lambda\omega$NK

Measure theory is the usual model for continuous[1] distributions. However, for our case classical measure theory has a critical shortcoming: function spaces of measure spaces are not necessarily measurable themselves, making measure theory unsuitable as the model of a programming language with λ-abstraction (Aumann et al., 1961). Put in a different way, the category of measurable spaces is not Cartesian closed.

Quasi-Borel Spaces (Heunen et al., 2017) are a recent advancement in the state-of-the-art of the semantics of probabilistic programs, which *are* Cartesian closed and provide an alternative formalisation of probabilistic structures. An even more recent development are the ω-Quasi-Borel Spaces (Vákár et al., 2019), which additionally provide ωCPO structure. We require this structure to model iterations.

The remainder of this chapter provides a brief overview of ω-Quasi-Borel Spaces, which is used as the semantic domain for the denotational semantics of P$\lambda\omega$NK that is presented in the next chapter. Eager readers may skip ahead to Chapter 8 on a first reading and come back when they want more detail.

[1] At first glance, it might seem counter-intuitive that P$\lambda\omega$NK could admit continuous probability distributions. However, Foster et al. (Foster et al., 2016) show an example of a PNK program where this is the case: let p be a program that outputs two distinct packets with equal probability, then p; (**dup**; p)* denotes a continuous distribution.

7.1 ω-Complete Partial Orders

Semantically modelling the behaviour of PλωNK or indeed plain PNK requires a semantic domain that captures the recursive nature of iterations C^*. For this purpose, the semantics is defined on a partially ordered domain and makes sure that the denotation increases with each iteration step, i.e., it ascends. The result of the entire iteration is then given by the least upper bound of the denotations of the iteration steps, and we must choose the ordering in such a way that these least upper bounds always exist. The ω-Complete Partial Orders (ωCPOs) are the orders with this property: in an ωCPO the least upper bounds of ascending chains always exist. Let us now define these concepts more precisely.

Definition 30. Let $\langle P, \sqsubseteq \rangle$ be some poset, an *ω-chain* is a sequence $(x_n)_{n\in\mathbb{N}}$ for $x_n \in P$, such that $\forall i, j \in \mathbb{N} : i \leq j$ implies $x_i \sqsubseteq x_j$. I will sometimes write such a chain as $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$.

Definition 31. The poset P is an *ω-Complete Partial Order (ωCPO)* when every ω-chain has a least upper bound (lub) $\bigvee_{n\geq 0} x_n \in P$. The *least upper bound* is the smallest element of P that is larger than every x_n. More formally,

$$\forall n \in \mathbb{N} : x_n \sqsubseteq \bigvee_{n\geq 0} x_n, \text{ and } \forall z \in P : (\forall n \in \mathbb{N} : x_n \subseteq z) \Rightarrow \bigvee_{n\geq 0} x_n \sqsubseteq z.$$

Example 32. The powerset 2^X of any set X is an ωCPO when ordered by subset inclusion ($\subseteq$). The lub of any ω-chain $X_0 \subseteq X_1 \subseteq \cdots$ (with $X_i \subseteq X$) is precisely the union $\bigcup_{i\geq 0} X_i$. In fact, $\langle 2^X, \subseteq \rangle$ is a complete lattice, meaning that any subset of 2^X has a lub.

Example 33. The functions $f : X \to P$ into an ωCPO $\langle P, \sqsubseteq_P \rangle$ also form an ωCPO, under the pointwise order $\preceq$, defined as $f \preceq g \iff \forall x \in X : f(x) \sqsubseteq_P g(x)$.

Continuous functions between two ωCPOs $\langle P, \sqsubseteq_P \rangle$ and $\langle Q, \sqsubseteq_Q \rangle$ are monotone functions $f : P \to Q$ such that f preserves least upper bounds, i.e. for all ω-chains $(a_n)_{n\in\mathbb{N}}$ in P,

$$f(\bigvee_{n\geq 0} x_n) = \bigvee_{n\geq 0} f(x_n).$$

Note that the first lub is in P, the second in Q. The monotonicity requirement ensures that this second lub actually exists, by ensuring that $(f(x_n))_{n\in\mathbb{N}}$ is an ω-chain.

The ωCPOs and continuous functions between them form a Cartesian closed category, i.e:

- the composition of two continuous functions is continuous;
- ωCPOs are closed under Cartesian products, i.e., given two ωCPOs P and Q, $P \times Q$ is also an ωCPO; and
- continuous functions between two ωCPOs also form an ωCPO. The order is the pointwise order.

7.2 Quasi-Borel Spaces

In P$\lambda\omega$NK we give semantics to higher-order functions and probabilistic choice. For this reason the semantic domain needs to be both Cartesian closed and admit a probabilistic power domain. As mentioned previously measure theory, the usual choice for probabilistic domains, is not suitable since it is not Cartesian closed. In contrast, Quasi-Borel Spaces(QBS) (Heunen et al., 2017) are suitable, since they are Cartesian closed.

However, in order to define QBSes, we must revisit some basic definitions from measure theory first. This treatment of measure theory is limited to the essentials needed to understand Quasi-Borel Spaces. For instance, a general discussion of measure spaces is avoided, by constraining ourselves to a particular set of measurable sets on the reals, the Borel sets:

Definition 34. The Borel sets $\mathfrak{B}$ are the least collection of subsets of $\mathbb{R}$, such that:

- the intervals $[a, b]$ are Borel sets for $a, b \in \mathbb{R}$,
- the complement of a Borel set is a Borel set, and
- countable unions of Borel sets are Borel sets.

A probability measure is a function $\mu : \mathfrak{B} \to [0, 1]$ satisfying $\mu(\mathbb{R}) = 1$ and $\mu(\bigcup S_n) = \sum \mu(U_n)$, for any countable sequence of disjoint Borel sets $(S_n)_{n\in\mathbb{N}}$. A function $f : \mathbb{R} \to \mathbb{R}$ is called *measurable* if its inverse image maps Borel sets to Borel sets. Symbolically, for all $B \in \mathfrak{B}$:

$$f^{-1}(B) = \{x \in \mathbb{R} \mid f(x) \in B\} \in \mathfrak{B}.$$

Such functions can be integrated with respect to a measure. For a *non-negative* real-valued measurable function $f : \mathbb{R} \to \mathbb{R}$, the integral of f with respect to a probability measure μ is defined as:

$$\int f \, d\mu = \sup_{(U_n)} \sum_n \left(\mu(U_n) \inf_{x \in U_n} f(x) \right)$$

where (U_n) ranges over finite partitionings of $\mathbb{R}$ into Borel sets. If f is allowed to be negative, its integral is:

$$\int f \, d\mu = \int f^+ \, d\mu - \int f^- \, d\mu$$

where $f^+ = \max(f, 0)$, $f^- = \max(-f, 0)$.

Measure theory generalises Borel sets to the measurable subsets of a measurable space X, and generalises measurable functions on $\mathbb{R}$ to measurable functions between measurable spaces, whose inverse image always maps measurable sets to measurable sets.

Quasi-Borel Spaces (Heunen et al., 2017) are an alternative to measure theory, starting from the from the notion of a random element $\mathbb{R} \to X$ instead of measurable sets.

Definition 35. A Quasi-Borel Space (QBS) $\langle X, M_X \rangle$ is a set X together with a set of functions M_X such that the following conditions are met:

- If $\alpha : \mathbb{R} \to X$ is constant, then $\alpha \in M_X$;
- if $\alpha \in M_X$, and $f : \mathbb{R} \to \mathbb{R}$ is measurable, then $\alpha \circ f \in M_X$;
- let $\mathbb{R} = \bigcup_{n \in \mathbb{N}} U_n$ where the U_n are pairwise disjoint Borel sets, if $\alpha_n \in M_X$ for all $n \in \mathbb{N}$, then $\beta \in M_X$ where $\beta(r) = \alpha_n(r)$ if $r \in U_n$.

Essentially, M_X must contain all constant functions, and must be closed under pre-composition with a measurable function or countable case-splitting. A function $f : X \to Y$ is a morphism from $\langle X, M_X \rangle$ to $\langle Y, M_Y \rangle$ if for all $\alpha \in M_X$, $f \circ \alpha \in M_Y$.

There are two canonical ways to turn a set X into a QBS. One option is to simply include *all* functions $\mathbb{R} \to X$ in M_X. Another option is to take as random elements all measurably piece-wise constant functions, i.e., those functions from $\mathbb{R}$ to X that are piece-wise constant on Borel sets.

The quasi-Borel spaces together with their morphisms form a category. Moreover, this category admits products, co-products and function spaces. That is, unlike the category formed by measurable spaces and measurable functions between them, this category is Cartesian closed (Heunen et al., 2017, Proposition 18).

7.3 ω-Quasi-Borel Spaces

Definition 36. An ω-Quasi-Borel Space is a triple $\langle X, M_X, \sqsubseteq_X \rangle$ such that:

- $\langle X, M_X \rangle$ is a quasi-Borel Space,
- $\langle X, \sqsubseteq_X \rangle$ is an ωCPO, and
- $\left(\bigvee_{n \geq 0} \alpha_n\right) \in M_X$ for all ω-chains $\alpha_0 \preceq \alpha_1 \preceq \cdots$ (with $\alpha_n \in M_X$), where $\preceq$ is the point-wise order on M_X.

Informally, an ωQBS is a QBS that is also an ωCPO, and whose random elements are closed under pointwise lubs of ω-chains.

The morphisms between ωQBSes are those morphisms between the underlying ωQBSes, which are also Scott-continuous between the underlying ωCPOs.

Let us reiterate two examples from Vákár et al. (2019):

Example 37. Real values have the ωQBS, $\langle \mathbb{R}, M_{\mathbb{R}}, = \rangle$, with $M_{\mathbb{R}}$ the set of measurable functions from $\mathbb{R}$ to $\mathbb{R}$. Alternatively, consider $\mathbb{W} = \langle [0, \infty], M_{\mathbb{W}}, < \rangle$ the space of weights, where $M_{\mathbb{W}}$ is the set of measurable functions from $\mathbb{R}$ to $[0, \infty]$.

Just like QBS, ωQBS forms a category that is closed under products, co-products and exponentials (functions). This category is Cartesian closed. Thus ωQBSes are a mathematical concept which at last unifies iteration, probabilistic choice and higher-order functions.

7.4 A Commutative Probabilistic Powerdomain

The denotational semantics for P$\lambda\omega$NK is a monadic semantics: it allows the structuring of the semantics in a compositional fashion (Moggi, 1991). This section explains how to define a monad suitable for expressing probabilistic computations as an ωQBS.

The idea, as explained by Vákár et al. (2019), is to treat distributions as expectation or integration operators. The distribution monad $D(X)$ is (a submonad[2] of) the continuation monad $C(X) = (X \to \mathbb{W}) \to \mathbb{W}$, where the arrows ($\to$) denote ωQBS morphisms. The idea is that for a $\mu \in C(X)$ and $f : X \to \mathbb{W}$, $\mu(f)$ computes the integral of f with respect to μ, in more traditional notation, $\mu(f) = \int f \, d\mu$.

The unit (*return*) and composition ($\gg\!=$) of this monad are given by:

$$return\ x = \lambda k \to k\ x \qquad\qquad m \gg\!= f = \lambda k \to m\ (\lambda x \to f\ x\ k)$$

[2]Specifically, it is the smallest ωQBS that is a full sub-ωCPO of C(X) and contains the randomisable random elements.

In more traditional language these represent integrating with δ_x (Dirac-delta of x) and the integral $\int_x \int k \ d(f \ x) \ dm$, respectively. The details of this construction are beyond the scope of this article. The most important results are:

- This construction satisfies the requirements of synthetic measure theory, implying that the results of measure theory continue to hold in this new setting. In particular, we can do addition and scalar multiplication of distributions.
- The monad consists of the *s-finite measures and kernels*, meaning that the monad is commutative: operations can be re-ordered (Staton, 2017).

Probabilistic powerdomains have been extensively investigated (see e.g., Saheb-Djahromi (1980); Jones and Plotkin (1989); Jung and Tix (1998); Goubault-Larrecq and Varacca (2011); Bacci et al. (2018); Battenfeld et al. (2007)). Nevertheless, until the work of Vákár et al. (2019), a convenient continuous probabilistic powerdomain that supports iteration and is commutative proved elusive.

However, the ω-quasi Borel Spaces are not the only approach to this problem. Dahlqvist and Kozen (2020) approach the subject from a completely different direction: rather than develop a novel formalisation of probability that is Cartesian closed, they abandon Cartesian closedness in favour of monoidal closedness, while retaining existing formalisations. It admits a natural definition of (Bayesian) conditioning.

Section 8.6 has some additional references.

Chapter 8

Denotational Semantics for P$\lambda\omega$NK

A denotational semantics maps terms (values and computations) into a semantic domain. A term's domain is determined by its type, therefore the semantics of types must be discussed first.

Figure 8.1 Denotations of types.

$$\begin{aligned}
[\![\mathbf{1}]\!] &= \langle \{\star\}, M_\star, = \rangle \\
[\![\mathcal{H}]\!] &= \langle \mathit{Headers}, M_{\mathit{Headers}}, = \rangle \\
[\![\mathbb{N}]\!] &= \langle \{1, \ldots, N\}, M_N, = \rangle \\
[\![T_1 \to T_2]\!] &= [\![T_1]\!] \to M([\![T_2]\!]) \\
A \rightharpoonup B &= A \to B_\perp \\
M(X) &= 2^{PH} \to D(X \rightharpoonup 2^{PH})
\end{aligned}$$

8.1 Semantics of Types

Figure 8.1 shows the interpretations of the different types. Each type denotes an ωQBS.

Unit types $\mathbf{1}$ are denoted by nullary products, whose only inhabitant is written as $\star$. Header types $\mathcal{H}$ are denoted by the finite set of header labels and header literals $\mathbb{N}$ are denoted by the natural numbers. In these three cases, the underlying ωCPO is given by the discrete order $(=)$[1] and the underlying ωQBS can be created using either of the canonical methods described in Section 7.2.

Function types $T_1 \to T_2$ are denoted by functions (morphisms) from the denotation of its argument $[\![T_1]\!]$ to $M([\![T_2]\!])$. Because this construction is standard for ωQBSes, the last two components of the triple are not written explicitly. Recall that the body of a function is a computation, producing side-effects: state, parallelism, and probabilistic choice. Therefore, the codomain of $[\![T_1 \to T_2]\!]$ is of the following form:

$$M(X) = \overbrace{2^{PH}}^{\text{input state}} \to D(\underbrace{X \rightharpoonup \overbrace{2^{PH}}^{\text{output state}}}_{\text{parallel value and state}})$$

where D is the distribution monad defined in Section 7.4, the arrow $(\to)$ denotes ωQBS-morphism and the harpoon $(\rightharpoonup)$ denotes partial maps. Partial maps $f : A \rightharpoonup B$ are equivalent to total functions $\widehat{f} : A \to B_\perp$, where $B_\perp$ is B extended with a distinguished element, $\perp_B$, preceding all other elements. For any $a \in A$, $\widehat{f}(a) = \perp_B$ then means that f is undefined on a. Hence, the domain $dom(f)$ of a partial map f is defined as $dom(f) = \{x \mid \widehat{f}(x) \neq \perp\}$. The partial maps are not required to be continuous (but they do form an ωCPO and an ωQBS under pointwise lubs).

The ωQBS of the state (sets of packet histories, 2^{PH}) is $\langle 2^{PH}, M_{2^{PH}}, \subseteq \rangle$ where $M_{2^{PH}}$ are the random elements of the exponential 2^{PH} of $\langle 2, M_2, \leq \rangle$ and $\langle PH, M_{PH}, = \rangle$.

Parallelism in both the value (X) and the state (2^{PH}) is modelled by partial maps $X \rightharpoonup 2^{PH}$. The idea is that a particular value comes with a particular state. Since the map is partial, not all values are necessarily present.

[1] In this order, everything is incomparable to everything, unless they are the same.

Furthermore, $M(X)$ is a monad with unit and composition given by:

$$return_M\ x = \lambda A \to return_D\ \lambda x \to A$$

$$m \gg\!=_{\mathrm{M}} f = \lambda A \to m\ A \gg\!=_{\mathrm{D}} \lambda \mu \to \Xi\{f\ x\ B \mid (x, B) \in \mu\}$$

where

$$m_1 \parallel_D\ m_2 = m_1 \gg\!=_{\mathrm{D}} \lambda \mu_1 \to m_2 \gg\!=_{\mathrm{D}} \lambda \mu_2 \to return_D\ (\mu_1 \bigvee \mu_2)$$

$$\Xi\{m_1, \cdots, m_n\} = m_1 \parallel_D \cdots \parallel_D\ m_2 \tag{8.1}$$

The $\gg\!=_{\mathrm{M}}$ operator first samples a map μ from the distribution mA, then applies f to every parallel value x and set B in the map, resulting in a set of distributions. The results are then aggregated using Ξ. This operator is defined in terms of the binary operator $\parallel_D$, which samples one map each from distributions m_1 and m_2, and returns the least upper bound. The use of Ξ is well-defined by the following lemma:

Lemma 38 (Lubs of finite sets)**.** *Let $\{m_1, \ldots, m_n\} \subseteq D(\llbracket T \rrbracket \rightharpoonup 2^{PH})$ for some type T, then $\Xi\{m_1, \ldots, m_n\}$ is the least upper bound of $\{m_1, \ldots, m_n\}$.*

The proof is a straightforward induction on the size of the set. Because Ξ computes the least upper bound, it is independent of the order of $\{m_1, \ldots, m_n\}$. However, it assumes that the input set is finite, i.e., that there are only finitely many unique m_i. This assumption is discharged by Theorem 39 (see Section 8.3).

It is also useful to lift $\parallel_D$ to M.

$$m_1 \parallel_M m_2 = \lambda A \to (m_1\ A \parallel_D\ m_2\ A)$$

Finally, the semantics $\llbracket T \rrbracket$ extends naturally to an interpretation on contexts (recall that contexts only contain values):

$$\llbracket x_1 : T_1, \ldots, x_k : T_k \rrbracket = \llbracket T_1 \rrbracket \times \cdots \times \llbracket T_k \rrbracket$$

That is, contexts are denoted by the k-ary product of the denotations of the types within the context. Again, because the construction is standard, the last two components of the triple are not written explicitly. For convenience, we can also write $\rho(x)$ to look up a variable x, and $[x \mapsto v]\rho$ to update the value of x in an environment $\rho \in \llbracket \Gamma \rrbracket$.

Figure 8.2 Denotations of terms.

Values
$[\![V]\!] : [\![\Gamma]\!] \to [\![T]\!]$
where $\Gamma \vdash_v V : T$

$$[\![x]\!]\ \rho = \rho(x)$$

$$[\![\mathbf{unit}]\!]\ \rho = \star$$

$$[\![h]\!]\ \rho = h$$

$$[\![i]\!]\ \rho = i$$

$$[\![\lambda x{:}T.C]\!]\ \rho =$$
$$\lambda v \to [\![C]\!]\ [x \mapsto v]\rho$$

Predicates
$[\![P]\!]^p : [\![\Gamma]\!] \to 2^{PH} \to 2^{PH}$
where $\Gamma \models \mathbf{wf}\ P$

$$[\![\mathbf{skip}]\!]^p\ \rho\ A = A$$

$$[\![\mathbf{drop}]\!]^p\ \rho\ A = \emptyset$$

$$[\![V_1 = V_2]\!]^p\ \rho\ A = \{\pi :: h \in A \mid \pi.([\![V_1]\!]\ \rho) = [\![V_2]\!]\ \rho\}$$

$$[\![\neg P]\!]^p\ \rho\ A = A - [\![P]\!]^p\ \rho\ A$$

$$[\![P_1 \wedge P_2]\!]^p\ \rho\ A = [\![P_1]\!]^p\ \rho\ A \cap [\![P_2]\!]^p\ \rho\ A$$

$$[\![P_1 \vee P_2]\!]^p\ \rho\ A = [\![P_1]\!]^p\ \rho\ A \cup [\![P_2]\!]^p\ \rho\ A$$

Computations $[\![C]\!] : [\![\Gamma]\!] \to M([\![T]\!])$ where $\Gamma \vdash_c C : T$

$$[\![\langle P \rangle]\!]\ \rho\ A = \mathit{return}_D\ (\lambda\star \to [\![P]\!]^p\ \rho\ A)$$

$$[\![V_1 \leftarrow V_2]\!]\ \rho\ A = \mathit{return}_D\ (\lambda\star \to \{\pi[[\![V_1]\!]\ \rho \mapsto [\![V_2]\!]\ \rho] :: h \mid \pi :: h \in A\})$$

$$[\![\mathbf{dup}]\!]\ \rho\ A = \mathit{return}_D\ (\lambda\star \to \{\pi :: \pi :: h \mid \pi :: h \in A\})$$

$$[\![C_1;\ C_2]\!]\ \rho\ = \mathbf{do}\{[\![C_1]\!]\ \rho; [\![C_2]\!]\ \rho\}$$

$$[\![C_1\ \&\ C_2]\!]\ \rho\ = [\![C_1]\!]\ \rho \parallel_M [\![C_2]\!]\ \rho$$

$$[\![C_1 \oplus_r C_2]\!]\ \rho\ = r([\![C_1]\!]\ \rho) + (1-r)([\![C_2]\!]\ \rho)$$

$$[\![C^*]\!]\ \rho\ A = \bigvee_{n\geq 0} ([\![C^n]\!]\ \rho\ A) \text{ where}$$
$$C^0 = \langle \mathbf{skip} \rangle,$$
$$C^{n+1} = \langle \mathbf{skip} \rangle\ \&(C;\ C^i), n \geq 0$$

$$[\![\mathbf{produce}\ V]\!]\ \rho\ = \mathit{return}_M\ ([\![V]\!]\ \rho)$$

$$[\![C_1\ \mathbf{to}\ x.C_2]\!]\ \rho\ = \mathbf{do}\{v \leftarrow [\![C_1]\!]\ \rho\ ; [\![C_2]\!]\ [x \mapsto v]\rho\ \}$$

$$[\![V_1\ V_2]\!]\ \rho\ = [\![V_1]\!]\ \rho\ ([\![V_2]\!]\ \rho)$$

Note: $X \to Y$ means ωQBS-morphisms from X to Y.

8.2 Semantics of Terms

Next, Figure 8.2 defines the denotational semantics for terms. The semantics are divided into three (mutually recursive) categories: semantics for values, for predicates and for other computations. All semantic functions take an environment $\rho \in [\![\Gamma]\!]$ as their first argument. Predicates and computations also take a set of packet histories $A \in 2^{PH}$.

Values For values, the semantics is relatively straightforward: variables can be looked up in the environment ρ. Unit, header and literal values map to their corresponding constants. An abstraction $\lambda x\!:\!S.C$ corresponds to a meta-function that takes a $v \in [\![T]\!]$, extends the environment ρ with v, and runs $[\![C]\!]$ in this new environment. To see why this makes sense, consider that it is the location where the function is defined that determines the scope of the *variables* in C, but it is the location where it is forced which determines its *state*.

Predicates Predicates filter sets of packet histories, that is, they allow or reject particular packet histories.

Predicates **skip** and **drop** allow, respectively disallow, all packet histories. A guard $V_1 = V_2$ only allows packet histories where the first packet's header $[\![V_1]\!]\ \rho$ has the specified value $[\![V_2]\!]\ \rho$ (accessing header f of a packet π is written as $\pi.f$). Negation ($\neg$) only retains packets that are dropped by its argument. Finally, conjunction ($\wedge$) and disjunction ($\vee$) take the intersection and union respectively.

Computations The semantics for computations returns values in the monad M. Since M is a monad, the text uses the well-understood *do-notation* to construct monadic expressions, de-sugaring straightforwardly to monadic bind ($\gg\!=_{\mathrm{M}}$).

The first three cases are atomic. Moreover, they all produce the unit value $\star$ and return a map from $\star$ to a modified set of histories.[2] Predicates P filter packet histories according to the predicate semantics $[\![P]\!]^p\ \rho\ A$. Modifications change the first packet's header for every packet history in the input set (setting a packet π's header f to x is written $\pi[f \mapsto x]$). Similarly, **dup** duplicates the first packet.

[2]By convention, λ with an arrow ($\rightarrow$) is abstraction in the meta-language (morphisms in ωQBS) and λ with dot . refers to abstraction in the object language.

Sequential composition $(C_1;\ C_2)$ first evaluates C_1, and then C_2 (returning the value of C_2). Since the evaluation of C_2 does not depend on the value of C_1, all packet histories of C_1 are simply aggregated and passed to C_2.

Parallel composition $C_1 \,\&\, C_2$ combines the results (values and states) of both computations using $\|_M$.

Choice $C_1 \oplus_p C_2$, reweighs both C_1 and C_2, by p and $1-p$ respectively, and adds the resulting distributions. Note that the sum is a probability distribution, i.e. all weights sum to one. Since D satisfies the requirements of synthetic measure theory, scalar multiplication and addition behave as one would intuitively expect, multiplying and adding probabilities.

Iteration (Kleene-star C^*) is defined as it is for PNK, using least fixed point semantics. It can be understood as the least fixed point of the function

$$\lambda m \to m \gg\!= \lambda \mu \to [\![\mathbf{skip} \,\&\, C]\!]\ \rho\ \mu(\star)$$

Operationally, it simply takes the least upper bound of iterating C for $0, 1, \ldots$ times. This definition goes at least as far back as (Kozen, 1981)'s early work on probabilistic program semantics.

To produce (**produce** V) a value V, simply return $[\![V]\!]\ \rho$ leaving the packet histories unchanged. For sequencing (C_1 **to** $x.C_2$) first obtain the value v produced by $[\![C_1]\!]\ \rho$, then extend the environment ρ with the value v, and pass this extended environment to $[\![C_2]\!]$. In the M-monad this corresponds to taking every unique value v and its set of packet histories that are produced by C_1 *in parallel* and passing them to C_2.

Application merely applies the denotation of V_1 (a meta-function) to the denotation of V_2.

Recall that $\gg\!=_{\mathrm{M}}$ is only well-defined for finite maps. This assumption is discharged by Theorem 39 (see Section 8.3). A further theorem (Theorem 42) makes the semantics well-defined. In particular, it ensures that the least upper bound in the definition of iteration exists. The next sections discuss each theorem in turn. The proofs of these theorems are in Appendix A.3.

8.3 Finite Maps

The function Ξ (8.1) is only defined on *finite* sets. Therefore, it must be verified that this function is never applied to an infinite set. Informally, this is the case if we ensure that in Figure 8.2, all maps inside M have a finite domain. More formally,

Theorem 39 (Finite Maps)**.** *For all computations C, such that $\Gamma \vdash_c C : T$, we have that for any $\rho \in [\![\Gamma]\!]$, $A \in 2^{PH}$, ωQBS morphism $f : X \to \mathbb{R}$, where $X = [\![T]\!] \multimap 2^{PH}$:*

$$\int_X f \, d([\![C]\!] \ \rho \ A) = \int_X \chi_F \, f \, d([\![C]\!] \ \rho \ A)$$

where $F = \{g \in X \mid dom(g) \text{ is finite}\}$, and $\chi_F : X \to \{0, 1\}$ is its characteristic function.

The proof of this theorem relies on logical relations (Tait, 1967) to define a stronger theorem (Theorem 41). The actual logical relations are $\mathcal{F}_v(T)$ and $\mathcal{F}_c(T)$, defined as follows:

Definition 40. The predicates $\mathcal{F}_v(T)$ and $\mathcal{F}_c(T)$ for types T are:

$$\mathcal{F}_v(\mathbf{1}) = [\![\mathbf{1}]\!] \qquad \mathcal{F}_v(\mathcal{H}) = [\![\mathcal{H}]\!] \qquad \mathcal{F}_v(\mathbb{N}) = [\![\mathbb{N}]\!]$$

$$\mathcal{F}_v(T_1 \to T_2) = \mathcal{F}_v(T_1) \to \mathcal{F}_c(T_2)$$

$$\mathcal{F}_c(T) = \left\{ f \in M([\![T]\!]) \;\middle|\; \begin{array}{l} \forall g \in G(T), \forall A \in 2^{PH} : \\ \\ \displaystyle\int g \, d(f(A)) = \int \chi_{F(T)} \, g \, d(f(A)) \end{array} \right\}$$

where

$$F(T) = \{\mu : \mathcal{F}_v(T) \multimap 2^{PH} \mid dom(\mu) \text{ is finite}\} \qquad G(T) = ([\![T]\!] \multimap 2^{PH}) \to \mathbb{R}$$

The idea of these logical relations is to restrict the denotations to those semantic objects where all partial maps have a finite domain. For instance, the denotation of non-function values never contains a map, hence $\mathcal{F}_v(T)$ is simply $[\![T]\!]$ in this case. Functions must preserve the finiteness of their arguments, hence $\mathcal{F}_v(T_1 \to T_2) = \mathcal{F}_v(T_1) \to \mathcal{F}_c(T_2)$.

In a somewhat roundabout fashion, $\mathcal{F}_c(T)$ says that the denotations of a computation of type T must only contain finite maps. In particular, it says that integrating any g with respect to $f(A)$ should be the same as integrating $\chi_{F(T)} \, g$ with respect to $f(A)$, where $F(T)$ only contains the finite maps of the appropriate type.[3] In other words, integrating while filtering out the finite maps, should not make a difference at all, meaning that the only maps that are

[3]The juxtaposition $\chi_{F(T)} \, g$ means pointwise multiplication.

present are finite. The following theorem, of which Theorem 39 is a corollary, states that every well-typed term's denotation is a member of the appropriate relation:

Theorem 41. *For all values V and computations C,*

- *if $\Gamma \vdash_v V : T$ then $[\![V]\!]\ \rho \in \mathcal{F}_v(T)$;*
- *if $\Gamma \vdash_c C : T$ then $[\![C]\!]\ \rho \in \mathcal{F}_c(T)$;*

where $\rho \in [\![\Gamma]\!]$ such that if $x : T_x \in \Gamma$, then $\rho(x) \in \mathcal{F}_v(T_x)$.

Proof. (sketch) By induction on the structure of the typing derivation, performing case analysis on the final rule application. The proof uses an equational reasoning style. See Appendix A.3.2. □

8.4 Continuity

Continuity is a property that intuitively means that a function preserves the least upper bounds of the domain it operates on. In other words, it preserves the ωCPO structure. This is a technical requirement to ensure that the least-upper bound of the semantics of iteration exists. More formally,

Theorem 42 (Continuity)**.** *For all computations C, such that $\Gamma \vdash_c C : T$, for all $\rho \in [\![\Gamma]\!]$, $[\![C]\!]\ \rho\ A$ is continuous in $A \in 2^{PH}$, i.e. $[\![C]\!]\ \rho\ A$ is monotone, and for all $A_1 \subseteq A_2 \subseteq \ldots \subseteq PH$:*

$$\bigvee_{i \geq 1} ([\![C]\!]\ \rho\ A_i) = [\![C]\!]\ \rho\ (\bigcup_{i \geq 1} A_i)$$

The proof of this theorem relies on the following logical relation:

Definition 43. The predicates $\mathcal{C}_v(T)$ and $\mathcal{C}_v(T)$ for types T are:

$$\mathcal{C}_v(\mathbf{1}) = [\![\mathbf{1}]\!] \qquad \mathcal{C}_v(\mathcal{H}) = [\![\mathcal{H}]\!] \qquad \mathcal{C}_v(\mathbb{N}) = [\![\mathbb{N}]\!]$$

$$\mathcal{C}_v(T_1 \to T_2) = \mathcal{C}_v(T_1) \to \mathcal{C}_c(T_2)$$

$$\mathcal{C}_c(T) = \left\{ f \in M([\![T]\!]) \;\middle|\; \begin{array}{l} f \text{ is continuous, and } \forall g \in G(T), \forall A \in 2^{PH} : \\ \int g\, d(f(A)) = \int \chi_{C(T)}\, g\, d(f(A)) \end{array} \right\}$$

where $C(T) = \{\mu : \mathcal{C}_v(T) \rightharpoonup 2^{PH}\}$, $G(T) = ([\![T]\!] \rightharpoonup 2^{PH}) \to \mathbb{R}$.

Similar to Section 8.3 the idea is to restrict the denotations to those semantics objects that are continuous (in the input set—not the environment), and for those partial maps that have a function domain, the domain only contains continuous functions. Note that the partial maps *themselves* are not required to be continuous. Indeed, this is clearly not the case (e.g. in the case of **produce** V). The denotations for non-function values do not contain input sets or maps, hence $\mathcal{C}_v(T)$ is simply $[\![T]\!]$. Functions can contain computations, thus $\mathcal{C}_v(T_1 \to T_2)$ are (not necessarily continuous) functions from $\mathcal{C}_v(T_1)$ to $\mathcal{C}_c(T_2)$. For computations of type T, $\mathcal{C}_c(T)$ says that the semantics itself must be continuous, and any element of a partial map's domain must be continuous.

We can now prove the following theorem, of which Theorem 42 is a direct consequence:

Theorem 44. *For all values V and computations C:*

- *if $\Gamma \vdash_v V : T$ then $[\![V]\!]\ \rho \in \mathcal{C}_v(T)$;*
- *if $\Gamma \vdash_c C : T$ then $[\![C]\!]\ \rho \in \mathcal{C}_c(T)$;*

where $\rho \in [\![\Gamma]\!]$ such that if $x : T_x \in \Gamma$, then $\rho(x) \in \mathcal{C}_v(T_x)$.

8.5 Conservativity

The next theorem relates the semantics to the original PNK semantics, showing that P$\lambda\omega$NK's semantics behaves identical to the original PNK semantics on probabilistic computations.

Theorem 45 (Conservativity)**.** *Let C be a closed probabilistic computation, and let $[\![C]\!]_{PNK}$ be the denotation of C in the probabilistic PNK semantics (Smolka et al., 2017b), re-translated into ωQBS (see Appendix A.2), then for all $A \in 2^{PH}$:*

$$[\![C]\!]_{PNK}\ A = [\![C]\!]\ \star\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \mathit{return}\ (\mu(\star))$$

Summary This chapter described a *well-defined* denotational semantics for P$\lambda\omega$NK. Well-definedness depends on two properties: finiteness of the partial maps and continuity. This semantics is a *conservative* extension of PNK's semantics. This chapter does not explain how this semantics can be evaluated. This is the subject of Chapter 9, which discusses how P$\lambda\omega$NK programs can be compiled to PNK in a semantics-preserving manner.

8.6 Related Work

Although PλωNK uses FG-CBV, not Call-By-Push-Value (CBPV), it was heavily inspired by it. CBPV was developed by Levy (2001) as a paradigm that subsumes both CBV and CBN. Since then, Ehrhard and Tasson (2019) have developed a probabilistic CBPV calculus. Technically, they give a semantics in terms of probabilistic coherence spaces (Danos and Ehrhard, 2011), whereas this work uses a monadic semantics based on ωQBSes.

Goubault-Larrecq (2019) cleverly side-steps the issue of providing a commutative statistical higher-order powerdomain, by giving a semantics for CBPV that interprets value types and computation types differently. The values are interpreted in a category which is closed under the powerdomain functor, and the computations are interpreted as DCPOs. His language also has demonic non-determinism, statistical termination testers and parallel if statements.

A crucial difference with this work is that monadic state is not present in either calculus, while it is PλωNK. This significantly complicates the denotational semantics, in particular for application and sequencing. A detailed investigation into the relationships between these calculi and PλωNK is reserved for future work.

Chapter 9

Compilation to PNK

Chapter 8 defined a denotational semantics for P$\lambda\omega$NK. This chapter explains how to compile a P$\lambda\omega$NK term into a corresponding PNK term in a way that preserves the semantics.

Compilation is made more difficult by the interaction between sequential and parallel composition. To understand the problem, let us expand the definition of the semantics of sequencing:

$$[\![C_1 \textbf{ to } x.C_2]\!]\ \rho\ A = [\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\,\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}$$

It uses every *unique* $(v, B) \in \mu$ exactly once, in parallel. Because parallel composition is not idempotent (Foster et al., 2016), duplicating (v, B) is not innocent. Therefore, the execution a program must ensure that it uses every unique (v, B) exactly once. To the best of my knowledge, there is no decidable procedure for this uniqueness problem. We could follow Smolka et al. (2017a) and remove **dup**, making the state-space finite and discrete. Although a perfectly valid solution, I believe that this too strong a restriction on the properties that can be modelled in the language (e.g., modelling multicast is not possible).

The alternative is to restrict the type of parallel composition to **1**, ensuring that the domain of μ contains a single element. This enables the compilation of closed P$\lambda\omega$NK terms into PNK. The program is then approximated with PNK's approximation procedure. The remainder of this chapter develops the latter approach.

For compilation, the new judgements $\Gamma \Vdash_v V : T$ and $\Gamma \Vdash_c C : T$ supplant $\Gamma \vdash_v V : T$ and $\Gamma \vdash_c C : T$. The rules for these judgements are analogous to the original rules, except that

$$\frac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 : T} \quad \text{changes into} \quad \frac{\Gamma \Vdash_c C_1 : \mathbf{1} \qquad \Gamma \Vdash_c C_2 : \mathbf{1}}{\Gamma \Vdash_c C_1 \,\&\, C_2 : \mathbf{1}}$$

The new rule restricts the type of parallel composition to **1**. In other words, the value of & is completely predictable: it must be **unit**. Furthermore, parallelism is the only way to grow the domain of the finite maps. Thus, all domains are now either a single function, or a single value. From a different perspective, we have just restricted the parallelism of P$\lambda\omega$NK to the parallelism present in PNK.

9.1 Elaboration

Only closed computations of type **1** can be compiled. This is reasonable because complete P$\lambda\omega$NK models are not functions and have no free variables. Initially, let us consider only *terminal* computations (Section 6.4), such computations are either **produce unit** or probabilistic computations (i.e. only consist of PNK terms). The following relation elaborates these computations into PNK:

Definition 46 (Elaboration)**.** Define $R \rightsquigarrow R$ as:

$$\begin{array}{rcll} R & \rightsquigarrow & R & \text{if } R \text{ is atomic} \\ R_1;\, R_2 & \rightsquigarrow & E_1;\, E_2 & \text{if } R_1 \rightsquigarrow E_1 \text{ and } R_2 \rightsquigarrow E_2 \\ R_1 \,\&\, R_2 & \rightsquigarrow & E_1 \,\&\, E_2 & \text{if } R_1 \rightsquigarrow E_1 \text{ and } R_2 \rightsquigarrow E_2 \\ R_1 \oplus R_2 & \rightsquigarrow & E_1 \oplus E_2 & \text{if } R_1 \rightsquigarrow E_1 \text{ and } R_2 \rightsquigarrow E_2 \\ R^* & \rightsquigarrow & E^* & \text{if } R \rightsquigarrow E \\ \textbf{produce } V & \rightsquigarrow & \textbf{skip} & \\ \lambda x{:}T.C & \rightsquigarrow & \textbf{skip} & \end{array}$$

Elaboration preserves the semantics of a closed term of type **1**:

Theorem 47 (Soundness of Elaboration)**.** *Let R_1, R_2 be terminals such that $\Vdash_c R_1 : \mathbf{1}$, $\Vdash_c R_2 : \mathbf{1}$ and $R_1 \rightsquigarrow R_2$, then $[\![R_1]\!] = [\![R_2]\!]$.*

Moreover, the elaboration always exists for closed terms of the right type:

✓**Theorem 48** (Completeness of Elaboration)**.** *Let R be a terminal, then there exists precisely one probabilistic E such that $R \rightsquigarrow E$.*

Figure 9.1 Rules for reduction from P$\lambda\omega$NK to PNK.

$$\boxed{C \Downarrow R}$$

$$\frac{}{\langle P\rangle \Downarrow \langle P\rangle} \qquad \frac{}{V_1 \leftarrow V_2 \Downarrow V_1 \leftarrow V_2} \qquad \frac{}{\mathbf{dup} \Downarrow \mathbf{dup}}$$

$$\frac{}{\mathbf{produce}\ V \Downarrow \mathbf{produce}\ V} \qquad \frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2}{C_1;\ C_2 \Downarrow R_1;\ R_2} \qquad \frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2}{C_1 \,\&\, C_2 \Downarrow R_1 \,\&\, R_2}$$

$$\frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2}{C_1 \oplus C_2 \Downarrow R_1 \oplus R_2} \qquad \frac{C \Downarrow R}{C^* \Downarrow R^*} \qquad \frac{[x \mapsto V]C \Downarrow R}{\lambda x{:}T.C\ V \Downarrow R}$$

$$\frac{C_1 \Downarrow \mathbf{produce}\ V \qquad [x \mapsto V]C_2 \Downarrow R}{C_1\ \mathbf{to}\ x.C_2 \Downarrow R} \qquad \frac{C_1 \Downarrow \langle P\rangle \qquad [x \mapsto \mathbf{unit}]C_2 \Downarrow R}{C_1\ \mathbf{to}\ x.C_2 \Downarrow \langle P\rangle;\ R}$$

$$\frac{C_1 \Downarrow F \leftarrow N \qquad [x \mapsto \mathbf{unit}]C_2 \Downarrow R}{C_1\ \mathbf{to}\ x.C_2 \Downarrow F \leftarrow N;\ R} \qquad \frac{C_1 \Downarrow \mathbf{dup} \qquad [x \mapsto \mathbf{unit}]C_2 \Downarrow R}{C_1\ \mathbf{to}\ x.C_2 \Downarrow \mathbf{dup};\ R}$$

$$\frac{C_1 \Downarrow {R_{11}}^* \qquad [x \mapsto \mathbf{unit}]C_2 \Downarrow R_2}{C_1\ \mathbf{to}\ x.C_2 \Downarrow {R_{11}}^*;\ R_2} \qquad \frac{C_1 \Downarrow R_{11};\ R_{12} \qquad R_{12}\ \mathbf{to}\ x.C_2 \Downarrow R_2}{C_1\ \mathbf{to}\ x.C_2 \Downarrow R_{11};\ R_2}$$

$$\frac{C_1 \Downarrow R_{11} \,\&\, R_{12} \qquad [x \mapsto \mathbf{unit}]C_2 \Downarrow R_2}{C_1\ \mathbf{to}\ x.C_2 \Downarrow (R_{21} \,\&\, R_{22});\ R_2}$$

$$\frac{C_1 \Downarrow R_{11} \oplus R_{12} \qquad R_{11}\ \mathbf{to}\ x.C_2 \Downarrow R_{21} \qquad R_{12}\ \mathbf{to}\ x.C_2 \Downarrow R_{22}}{C_1\ \mathbf{to}\ x.C_2 \Downarrow R_{21} \oplus R_{22}}$$

9.2 Reduction

Converting a closed terminal into a PNK program is only half the battle. The other half is performed by the bigstep relation $C \Downarrow R$ given in Figure 9.1. It reduces a computation to a terminal. The following observations hold about the bigstep relation:

- Atomic computations simply reduce to themselves.
- Compound computations built with sequential and parallel composition, probabilistic choice and iteration are reduced to the reduction of their subcomputations.

- Application of a lambda follows the usual rule, substituting the value into the body of the lambda.
- When C_1 reduces to **produce** V, sequencing (C_1 **to** $x.C_2$) substitutes V for x in C_2, reducing the result of the substitution. When the first argument instead reduces to a predicate, a modification, a duplication, parallel composition or an iteration, C_1 always reduces to a terminal of type **1**. Hence, **unit** is substituted for V in these cases. When the first argument reduces to sequential composition composition or probabilistic choice, sequencing, like application also *distributes* over this argument. Finally, if the first argument reduces to parallel composition, it must produce a unit value, and so we can always substitute **unit** for x.

To be clear, the compilation strategy is as follows: (1) *Reduce* a P$\lambda\omega$NK computation to a terminal, and (2) *elaborate* the remaining terminal into PNK. To ensure that the compilation delivers correct results, it remains to show that step (1) terminates, and that this step is sound. Soundness means that the denotational semantics of the program is preserved:

Theorem 49 (Soundness of Reduction)**.** *Let C, R be computations, if $\Gamma \Vdash_c C : T$ and $C \Downarrow R$ then $[\![C]\!] = [\![R]\!]$.*

Termination is the subsequent concern. It follows from strong normalisation of the reduction relation, a property of the meta-theory of reduction, discussed in the next section.

9.3 Meta-theory of Reduction

The reduction relation obeys the standard type-preservation theorems, and in addition, is *strongly normalising*. In detail, terminals are reduced to themselves (Theorem 50). Moreover, all reductions result in terminals (by definition), in a deterministic fashion (Theorem 51) and preserve types (Theorem 52). Finally, reduction is strongly normalising (Theorem 53).

✓**Theorem 50** (Reflection)**.** *Let R be a terminal, then $R \Downarrow R$.*

✓**Theorem 51** (Determinacy)**.** *Let C, R_1, R_2 be computations, if $C \Downarrow R_1$ and $C \Downarrow R_2$, then $R_1 = R_2$.*

✓**Theorem 52** (Preservation)**.** *Let C, R be computations, if $\Gamma \Vdash_c C : T$ and $C \Downarrow R$, then $\Gamma \vdash_c R : T$. Conversely, if $\Gamma \Vdash_c C : T_1$, $\Gamma \Vdash_c R : T_2$ and $C \Downarrow R$, then $T_1 = T_2$.*

Figure 9.2 Logical relations involved in proving Strong Normalisation, for values, terminals and computations, indexed by type.

$$\mathbf{V}[\mathbf{1}] = \{unit\} \quad \mathbf{V}[\mathcal{H}] = \mathit{Headers} \quad \mathbf{V}[\mathbb{N}] = \mathbb{N}$$
$$V_1 \in \mathbf{V}[T_1 \to T_2] \text{ iff } \Vdash_v V_1 : T_1 \to T_2 \text{ and } \forall V_2 \in \mathbf{V}[T_1] : V_1\ V_2 \in \mathbf{C}[T_2]$$

$$R \in \mathbf{T}[\mathbf{1}] \text{ iff } \Vdash_c R : \mathbf{1} \text{ where } R \text{ is atomic}$$
$$R^* \in \mathbf{T}[\mathbf{1}] \text{ iff } \Vdash_c R^* : \mathbf{1} \text{ and } R \in \mathbf{T}[\mathbf{1}]$$
$$\mathbf{produce}\ V \in \mathbf{T}[T] \text{ iff } \Vdash_c \mathbf{produce}\ V : T \text{ and } V \in \mathbf{V}[T]$$
$$R_1;\ R_2 \in \mathbf{T}[T] \text{ iff } \Vdash_c R_1;\ R_2 : T \text{ and } \exists T', R_1 \in \mathbf{T}[T'] \text{ and } R_2 \in \mathbf{T}[T]$$
$$R_1 \,\&\, R_2 \in \mathbf{T}[T] \text{ iff } \Vdash_c R_1 \,\&\, R_2 : T \text{ and } R_1, R_2 \in \mathbf{T}[T]$$
$$R_1 \oplus R_2 \in \mathbf{T}[T] \text{ iff } \Vdash_c R_1 \oplus R_2 : T \text{ and } R_1, R_2 \in \mathbf{T}[T]$$

$$C \in \mathbf{C}[T] \text{ iff } \Vdash_c C : T \text{ and } \exists R \in \mathbf{T}[T] : C \Downarrow R$$

✓**Theorem 53** (Strong Normalisation)**.** *Let C be a computation such that $\Vdash_c C : T$ for some computation type T, then there exists a terminal R such that $C \Downarrow R$.*

In addition to the theorems shown here, P$\lambda\omega$NK also satisfies additional inversion and substitution lemmas that are instrumental in proving these theorems.

The meta-theory discussed in this section has been mechanised with the aid of the Abella proof-assistant (Gacek, 2008). The proofs for these theorems proceed by induction either on the structure of terminals or the structure of the bigstep relation. Each of proof has many cases that need to be checked. Using a theorem prover, ensures that no cases or conditions are forgotten.

The most involved proof is Strong Normalisation, which requires a logical-relation style proof technique (Tait, 1967). This particular proof was inspired by the standard proof of strong normalisation of CBPV, described in Levy's thesis (Levy, 2001).

The definition of the logical relations are shown in Figure 9.2. In essence, we need to define three mutually-recursive type-indexed logical relations: one for values ($\mathbf{V}[T]$), one for closed *terminal* computations ($\mathbf{T}[T]$) and one for all computations, terminal or non-terminal ($\mathbf{C}[T]$). The idea is that $\mathbf{C}[T]$ contains only closed computations for which the bigstep relation $\Downarrow$ terminates, and only produces terminating values. For values, $\mathbf{V}[T]$ contains all closed non-function values, and only closed values of function type that preserve termination when applied to terminating values.

Compared to Levy's logical relations, this logical relation does not contain product or sum types. The proof leverages standard Abella techniques to prove theorems with logical relations, which requires defining and proving substitution lemmas for every syntactic form. Moreover, it needs several additional preservation lemmas for sequential and parallel composition, and for probabilistic choice.

The proof of strong normalisation, together with the definitions of the logical relations and supporting lemmas amounts to a little under 800 lines of Abella code (roughly 1500 LOC in total).

9.4 Discussion

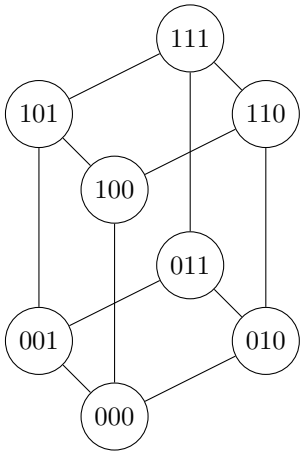
This chapter identifies a class of P$\lambda\omega$NK programs that can be safely compiled to PNK. In particular, the class consists of computations C such that $\Vdash_c C : \mathbf{1}$, i.e., those programs that do not use parallelism beyond what is present in PNK. It should be possible to ease this restriction slightly. For instance, unlike functions, distinct headers and literals can be distinguished. This leads to only finitely many cases, which could be encoded explicitly into PNK.

Once a program has been compiled, it can be approximated in PNK. The approximation proceeds by expanding iterations in the program up to n times, for finite n. More iterations improve the accuracy of the results (Smolka et al., 2017b).

9.5 Example

Let us demonstrate the expressivity of P$\lambda\omega$NK by revisiting an example from earlier work PNK (Foster et al., 2016). The compilation procedure fully reduces this program to a PNK program.

Figure 9.3 Gossip Network



(a) Topology

Round	Expected # Infections
0	1
1	2
2	3.333333333333333
3	4.864197530864196
4	6.245084590763608
5	7.172046065947742
6	7.657586571266758
7	7.871210010612862

(b) Expected Infections

Example 54 (Gossip Protocols)**.** In a gossip network, information is disseminated by gossiping, i.e. randomly spreading information to one's direct neighbours, who in turn divulge this information to their neighbours, and so on. Of interest is the expected number of nodes that have been infected by the gossip, after a given number of rounds.

The example borrows a situation from Foster et al.'s paper (Foster et al., 2016), where 8 nodes are organised in a hypercube (each node is connected to 3 other nodes, see Figure 9.3a). The nodes are addressed with 3-bit binary numbers. To simplify routing, addresses of neighbouring nodes differ in exactly one bit, for instance, node 000 is connected to nodes 100, 010 and 001.

In PNK this example requires an informal meta-construction, facilitating iteration over multiple nodes. In P$\lambda\omega$NK, iteration is modelled directly with higher-order functions.

P$\lambda\omega$NK doesn't support operations on binary numbers directly. The program builds them from Church-encoded bits instead:

$$
\begin{array}{l}
\text{// single bits} \\
\mathit{zero} \quad = \lambda x.\lambda y.\mathbf{produce}\ x \\
\mathit{one} \quad = \lambda x.\lambda y.\mathbf{produce}\ y \\
\mathit{forallBit} = \lambda f.f\,\mathit{zero}\,\&\,f\,\mathit{one} \\
\\
\mathit{flip} \quad = \lambda b.\lambda x.\lambda y.\mathit{byx} \\
\\
\text{// 3-bit addresses} \\
\mathit{toValue} \ = \lambda b_2.\lambda b_1.\lambda b_0. \\
\qquad\quad b_0\ 0\ 1\ \mathbf{to}\ N_0^1.b_0\ 2\ 3\ \mathbf{to}\ N_2^3.b_0\ 4\ 5\ \mathbf{to}\ N_5^4.b_0\ 6\ 7\ \mathbf{to}\ N_6^7. \\
\qquad\quad b_1\ N_0^1\ N_2^3\ \mathbf{to}\ N_0^3.b_1\ N_4^5\ N_6^7\ \mathbf{to}\ N_4^7. \\
\qquad\quad b_2\ N_0^3\ N_4^7 \\
\\
\mathit{forall} \quad = \lambda f.\mathit{forallBit}\ (\lambda b_2.\mathit{forallBit}\ (\lambda b_1.\mathit{forallBit}\ (\lambda b_0.f\ b_2\ b_1\ b_0)))
\end{array}
$$

The function *toValue* converts a binary number back into a P$\lambda\omega$NK field, and *forall* executes a function for all addresses.

The hypercube topology of the network is defined as follows:

$$
\begin{array}{ll}
\mathit{send} & = \lambda \mathit{src}.\lambda \mathit{dst}.(SW = \mathit{src};\ PT = \mathit{dst}) \\
\mathit{recv} & = \lambda \mathit{src}.\lambda \mathit{dst}.(SW \leftarrow \mathit{dst};\ PT \leftarrow 0) \\
\mathit{link} & = \lambda \mathit{src}.\lambda \mathit{dst}.(\mathit{send}\ \mathit{src}\ \mathit{dst};\ \mathit{recv}\ \mathit{src}\ \mathit{dst}) \\
\mathit{linkn} & = \lambda sb_2.\lambda sb_1.\lambda sb_0.\lambda db_2.\lambda db_1.\lambda db_0. \\
 & \quad \mathit{toValue}\ sb_2\ sb_1\ sb_0\ \mathbf{to}\ \mathit{src}. \\
 & \quad \mathit{toValue}\ db_2\ db_1\ db_0\ \mathbf{to}\ \mathit{dst}. \\
 & \quad \mathit{link}\ \mathit{src}\ \mathit{dst} \\
\mathit{neighboursD} & = \lambda \mathit{act}.\lambda b_2.\lambda b_1.\lambda b_0. \\
 & \quad \mathit{flip}\ b_0\ \mathbf{to}\ b_0.\mathit{act}\ b_2\ b_1\ b_0 \\
 & \&\ \mathit{flip}\ b_1\ \mathbf{to}\ b_1.\mathit{act}\ b_2\ b_1\ b_0 \\
 & \&\ \mathit{flip}\ b_2\ \mathbf{to}\ b_2.\mathit{act}\ b_2\ b_1\ b_0 \\
\mathit{hypert} & = \mathit{forall}\ (\lambda b_2.\lambda b_1.\lambda b_0. \\
 & \&\ \mathit{linkn}\ b_2\ b_1\ b_0\ b_2\ b_1\ b_0 \\
 & \&\ \mathit{linkn}\ b_2\ b_1\ b_0\ \mathbf{to}\ \mathit{act}.\mathit{neighboursD}\ \mathit{act}\ b_2\ b_1\ b_0)
\end{array}
$$

The following program specifies the randomised routing (information dissemination):

$$
\begin{array}{ll}
\mathit{forward} & = \lambda \mathit{src}.\mathit{SW} = \mathit{src} \\
\mathit{tow} & = \lambda \mathit{dst}.\mathit{PT} \leftarrow \mathit{dst} \\
\mathit{checksw} & = \lambda b_2.\lambda b_1.\lambda b_0.\mathit{toValue}\ b_2\ b_1\ b_0\ \mathbf{to}\ n.\mathit{forward}\ n \\
\mathit{setpt} & = \lambda b_2.\lambda b_1.\lambda b_0.\mathit{toValue}\ b_2\ b_1\ b_0\ \mathbf{to}\ n.\mathit{tow}\ n \\
\mathit{neighboursC} & = \lambda \mathit{act}.\lambda b_2.\lambda b_1.\lambda b_0. \\
 & \quad (\mathit{flip}\ b_0\ \mathbf{to}\ b_0.\mathit{act}\ b_2\ b_1\ b_0) \oplus_{\frac{1}{3}} \\
 & \quad (\mathit{flip}\ b_1\ \mathbf{to}\ b_1.\mathit{act}\ b_2\ b_1\ b_0 \oplus_{\frac{1}{2}} \mathit{flip}\ b_2\ \mathbf{to}\ b_2.\mathit{act}\ b_2\ b_1\ b_0) \\
\mathit{switch} & = \lambda b2.\lambda b1.\lambda b0. \\
 & \quad \mathit{checksw}\ b_2\ b_1\ b_0;\ (\mathit{neighboursC}\ \mathit{setpt}\ b_2\ b_1\ b_0\ \&\ \mathit{setpt}\ b_2\ b_1\ b_0)) \\
p & = \mathit{forall}\ \mathit{switch}
\end{array}
$$

Note that each node also re-infects itself. Finally, the main predicate ties it all together:

$$\mathit{SW} \leftarrow 0; \mathit{PT} \leftarrow 0;\ (p;\ \mathit{hypert})^*$$

The table in Figure 9.3b shows the expected number of infected nodes after each round, as computed by running the model. Note that the expected value approaches eight (i.e., all nodes become infected) as the number of rounds increases.

Chapter 10

Conclusion

> *“ If you’re going to do this damned silly thing, don’t do it in this damned silly way. ”*
>
> — *Sir Humphrey Appleby, Yes Minister*

10.1 Goal of the Thesis

Accidental complexity is complexity that stems not from the problem itself, but from solving it on a computer, e.g., memory management, updating loop indices, synchronising concurrent threads, etc. Accidental complexity causes software defects, either directly, or indirectly, because time wasted on accidental complexity cannot be devoted to the real problem instead. Therefore, if we are to build reliable software—on which our society is ever more dependent—we must avoid accidental complexity wherever possible.

Contrary to imperative programming, declarative programming avoids accidental complexity by not caring how a problem is solved: it defines *what* the problem is, rather than *how* the problem is solved. Thus, accidental complexity is averted by not expressing the operational issues that cause it in the first place.

The next sections discuss to how this thesis has contributed to declarative programming in logic, functional, and probabilistic programming.

10.2 Summary of Part I

Part I concerns tabling, in particular sound answer subsumption for Prolog, and monadic tabling in Haskell.

Chapter 2 Introduction to Tabling

This chapter mainly recalls the motivation and theoretical background for tabling. Most of the material in this chapter is well-known in the literature.

The background focuses on the interpretation of a logic program as its set of true atoms, its *least Herbrand model*, captured by the least fixed point of the monotone T_P operator. Other semantics for logic programming, such as SLD-resolution (Kowalski, 1974) are operational, and therefore not conductive to a declarative understanding of logic programming.

Because tabled logic programming aligns more closely with this interpretation in terms of a least Herbrand model, tabling is inherently more declarative than traditional SLD-resolution (Kowalski, 1974).

Chapter 3 Sound Answer Subsumption

In many instances of tabling only the optimal answers to a query are relevant. To improve performance over a naive generate-and-aggregate approach, various forms of answer subsumption that greedily combine these answers have been developed in the literature. However, their semantics has never been described formally. An operational understanding is always an option in this case, and although often useful, it is a far cry from the declarative ideal that tabling promises.

Chapter 3 defines a high-level semantics for answer subsumption based on lattice theory, which is then generalised to establish a correctness condition indicating when it is safe to use (greedy) answer subsumption. The chapter contains several examples where the existing implementations of answer subsumption fail that condition and derive an erroneous result.

Chapter 4 Monadic Tabling in Haskell

Chapter 4 describes mutually recursive non-deterministic computations in a free monadic fashion, and gives these descriptions a concise denotational semantics in terms of sets, and more generally in terms of lattices. These semantics are

implemented efficiently in Haskell using the Effect Handlers approach. The implementation is validated by declaratively implementing several dynamic programming problems.

10.3 Summary of Part II

Part II presents the probabilistic modelling language P$\lambda\omega$NK.

Chapter 5 Probabilistic Network Modelling

This chapter introduces the motivation for probabilistic modelling of computer networks in general, and the need for a (higher-order) *functional* probabilistic modelling language in particular.

Chapter 6 Functional Probabilistic NetKAT

After a brief introduction to modelling in Probabilistic NetKAT (PNK), this chapter presents the syntax and type system of P$\lambda\omega$NK, a higher-order functional extension of PNK. The type-system is strongly normalising. Since the main purpose of P$\lambda\omega$NK is verification, the additional flexibility of general recursion is not required. Indeed, Chapter 8 shows that strong normalisation is necessary to make the semantics well-defined. Moreover, Chapter 9 develops a procedure to compile programs in P$\lambda\omega$NK to PNK, given small type restrictions.

Chapter 7 A Convenient Category for P$\lambda\omega$NK

Because P$\lambda\omega$NK combines higher-order functions and probability, it cannot have a purely measure-theoretic semantics. Instead, Chapter 8 leverages ω-Quasi Borel Spaces to define a denotational semantics. Chapter 7 reviews the necessary background on measure theory, Quasi Borel Spaces and, ω-Quasi Borel Spaces.

Chapter 8 Denotational Semantics for P$\lambda\omega$NK

Chapter 8 defines the denotational semantics for P$\lambda\omega$NK that combines state, non-determinism and probability distributions. This semantics is conservative

with respect to the original semantics for PNK (Theorem 45). The well-definedness of the semantics is proven in several theorems (Theorems 39 and 42). These proofs rely on logical relations that are interesting in their own right.

Chapter 9 Compilation to P$\lambda\omega$NK

Finally, Chapter 9 develops a procedure to compile programs in P$\lambda\omega$NK to the simpler language Probabilistic NetKAT, given that the type of parallel composition is restricted to computations of the unit type. The termination of the compilation procedure is a consequence of the strong normalisation of the type system. Strong normalisation and other meta-theoretical properties were proved with the aid of the Abella proof assistant.

10.4 Future Work

Sound Answer Subsumption

The correctness condition (3.4) is sufficient, but not necessary: there may still exist programs that do not satisfy the condition, for which the greedy strategy nevertheless delivers correct results. Since I have not run across any non-contrived examples of such programs, I contend that this apparent lack of necessity is an artefact of my rather coarse semantics, which I intend to refine in future work.

The verification of correctness does constitute a non-trivial effort. Hence, manually proving the correctness condition for realistically sized programs could be unfeasible in practice. In the future we would ideally have an automated analysis that warns the programmer if it fails to establish the correctness condition. A light-weight approach, similar to the type system used by Arntzenius and Krishnaswami (2020) for Datafun, is preferable.

Monadic Tabling

Some programs do not require the full generality of a monadic syntax. In those cases, it might be beneficial to restrict tabled computations to applicative (McBride and Paterson, 2008) or selective applicative functors (Mokhov et al., 2019). The advantage is that for such computations the dependency graph of the tabled calls can be statically approximated, instead of dynamically discovered (as is necessary for monads). In theory, dynamic discovery can lead

to duplications when a value from a recursive call does not change the structure of its continuation, when the continuation is applicative, the program's entire structure is static, and the continuation structure is obvious.

The flexibility of the effect handler approach comes at the cost of a significant performance overhead. Recent work on fusion for effects (Wu and Schrijvers, 2015) could be leveraged to reduce this overhead.

P$\lambda\omega$NK

Future work should aim to quantitatively evaluate the impact of P$\lambda\omega$NK, by re-implementing the existing PNK case-studies, and study the improvements in terms of readability and maintainability. To support this quantitative study, the prototype needs to be developed and optimised further. I believe that performance can be improved by incorporating techniques such as knowledge compilation (Smolka et al., 2015; Kisa et al., 2014) and defunctionalisation (Reynolds, 1998; Danvy and Nielsen, 2001).

Furthermore, this work contains a significant amount of paper proofs. These proofs are inductive proofs, using equational reasoning and logical relations, which should not be too difficult to mechanise. However, the requisite background theory, i.e., (ω-)Quasi-Borel Spaces, has to be mechanised first.

10.5 Concluding Remarks

Tabling improves the declarativity of logic programs. In addition, answer subsumption makes it possible to write greedy algorithms for optimisation problems in a declarative way, provided that the (declarative) correctness condition (3.4) is satisfied. In the same way, monadic tabling in Haskell fixes non-determinism and recursion to align with a more useful, declarative meaning.

PNK's focus is already on declarative modelling of networks, abstracting away hardware and protocol details. Yet, PNK introduces new accidental complexity by not having any abstraction facilities, something I have sought to remedy in this work.

In conclusion, this thesis has contributed towards a more declarative and less error-prone future for our software-driven world.

Appendices

Appendix A

Additional material for P$\lambda\omega$NK

A.1 Atomic Computations

The denotational semantics predicates, modifications and duplications all share a very similar structure. Essentially, they immediately return a partial map from the unit value to a set, that is somehow a transformation of the input set. These are called *atomic* computations (see Smolka et al. (2017b)).

This property is very helpful, since, for the purpose of proving properties about their denotational semantics, atomic computations can be treated uniformly.

To make this more precise, we first define how individual elements of the input set are transformed. For predicates, there is a boolean function B_P that decides if an element of the input is retained in the output:

Definition 55. Let P be a predicate such that $\Gamma \vdash_c P : \mathbf{1}$, then $B_P : [\![\Gamma]\!] \to PH \to \{0, 1\}$ is defined as:

$$
\begin{aligned}
B_{\mathbf{skip}}\ \rho\ x &= 1 \\
B_{\mathbf{drop}}\ \rho\ x &= 0 \\
B_{V_1=V_2}\ \rho\ (\pi :: h) &= \pi.[\![V_1]\!]\ \rho = [\![V_2]\!]\ \rho \\
B_{P_1 \wedge P_2}\ \rho\ x &= B_{P_1}\ \rho\ x \wedge B_{P_2}\ \rho\ x \\
B_{P_1 \vee P_2}\ \rho\ x &= B_{P_1}\ \rho\ x \vee B_{P_2}\ \rho\ x \\
B_{\neg P}\ \rho\ x &= \neg(B_P\ \rho\ x)
\end{aligned}
$$

Lemma 56. *Let P e a predicate such that $\Gamma \vdash_c \langle P \rangle : \mathbf{1}$, $\rho \in [\![\Gamma]\!]$ and $A \in 2^{PH}$: then*

$$[\![p]\!]^p\ \rho\ A = \{x \in A \mid B_P\ \rho\ x\}$$

Proof. By straightforward induction on the structure of the typing derivation. □

For atomic computations there is a function f_P that transforms the input elements:

Definition 57. Let C be an atomic computation, such that $\Gamma \vdash_c C : \mathbf{1}$ then $f_C : [\![\Gamma]\!] \to PH \rightharpoonup PH$ is defined as:

$$
\begin{aligned}
f_P\ \rho\ x &= \begin{cases} x & \text{if } B_P\ \rho\ x \\ \bot & \text{otherwise} \end{cases} \\
f_{V_1 \leftarrow V_2}\ \rho\ (\pi :: h) &= \pi[[\![V_1]\!]\ \rho \mapsto [\![V_2]\!]\ \rho] :: h \\
f_{\mathbf{dup}}\ \rho\ (\pi :: h) &= \pi :: \pi :: h
\end{aligned}
$$

Lemma 58. *Let C be an atomic computation such that $\Gamma \vdash_c C : \mathbf{1}$, then $[\![C]\!]\ \rho\ A = return_D\ (\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})$ for all $\rho \in [\![\Gamma]\!]$ and $A \in 2^{PH}$.*

Here it should be understood that if $f_C\ \rho$ is undefined on some element x, the element is not included in the resulting set.

Proof. By case analysis on the structure of the atomic computation C. If C is a predicate, the required follows from the definition of $[\![C]\!]\ \rho\ A$ and Lemma 56. If C is a modification or duplication, the required follows immediately from the definition. □

A.2 Re-translation of PNK semantics

Smolka et al. (2017b) give denotational semantics for deterministic and probabilistic PNK programs at the same time, by interpreting the same semantics in a different monad (the identity monad and the monad of (sub)probability measures, respectively). We only concern ourselves with the probabilistic semantics.

First note that they do not distinguish between $\wedge$ and ; or $\vee$ and &, which we do. For this reason we define $[\![\cdot]\!]_{PNK}$ as:

$$
\begin{aligned}
[\![C]\!]_{PNK} &: 2^{PH} \to D(2^{PH}) \\
[\![\mathbf{drop}]\!]_{PNK}\ A &= \mathit{return}_D\ \emptyset \\
[\![\mathbf{skip}]\!]_{PNK}\ A &= \mathit{return}_D\ A \\
[\![f = n]\!]_{PNK}\ A &= \mathit{return}_D\ (\{\pi :: h \in A \mid \pi.f = n\}) \\
[\![\neg P]\!]_{PNK}\ A &= [\![P]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda B \to \mathit{return}_D\ (B - A) \\
[\![P_1 \wedge P_2]\!]_{PNK}\ A &= [\![P_1]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} [\![P_2]\!]_{PNK} \\
[\![P_1 \vee P_2]\!]_{PNK}\ A &= [\![P_1]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda B \to \\
&\quad\ [\![P_2]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda C \to \\
&\quad\ \mathit{return}_D\ (B \cup C) \\
[\![f \leftarrow n]\!]_{PNK}\ A &= \mathit{return}_D\ (\{\pi[f \mapsto n] :: h \in A \mid \pi :: h \in A\}) \\
[\![\mathbf{dup}]\!]_{PNK}\ A &= \mathit{return}_D\ (\{\pi :: \pi :: h \in A \mid \pi :: h \in A\}) \\
[\![C_1;\ C_2]\!]_{PNK}\ A &= [\![C_1]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} [\![C_2]\!]_{PNK} \\
[\![C_1\ \&\ C_2]\!]_{PNK}\ A &= [\![C_1]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda B \to \\
&\quad\ [\![C_2]\!]_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda C \to \\
&\quad\ \mathit{return}_D\ (B \cup C) \\
[\![C_1 \oplus_r C_2]\!]_{PNK}\ A &= r([\![P_1]\!]\ A\ ) + (1 - r)([\![P_2]\!]\ A\ ) \\
[\![C^*]\!]_{PNK}\ A &= \bigsqcup_{n \geq 0} [\![C^n]\!]_{PNK}\ A
\end{aligned}
$$

A.3 Proofs

A.3.1 Lemma 38

Proof. Let $M = \{m_1, \ldots, m_n\}$. By induction on n, we show that ΞM is the least upper bound of M.

Case BASE $n = 0$
If $n = 0$, then $M = \emptyset$ and the statement is vacuously true.

Case INDUCTION $n + 1$
Observe that:

$$\Xi M = \Xi\{m_1, \ldots, m_n, m_{n+1}\} = \Xi\{m_1, \ldots, m_n\} \parallel_M m_{n+1}$$

By induction, we know that $\Xi\{m_1, \ldots, m_n\}$ is the least upper bound of $\{m_1, \ldots, m_n\}$. Hence, we must only show that for any $m_1, m_2 \in D(X)$, $m_1 \parallel_M m_2$ is the least upper bound of m_1 and m_2. Recall that D is a continuation monad, then we can proceed as follows (except where otherwise noted, the proof proceeds by continuity):

$$\begin{aligned}
& m_1 \parallel_M m_2 \\
= \;& m_1 \gg\!= \lambda x \to m_2 \gg\!= \lambda y \to \mathit{return}\ x \bigvee y \\
= \;& \lambda k \to m_1\ (\lambda x \to m_2\ (\lambda y \to k\ (x \bigvee y))) \\
= \;& \lambda k \to m_1\ (\lambda x \to m_2\ (\lambda y \to k\ x \bigvee k\ y)) \\
= \;& \lambda k \to m_1\ (\lambda x \to m_2\ ((\lambda y \to k\ x) \bigvee (\lambda y \to k\ y))) \\
= \;& \lambda k \to m_1\ (\lambda x \to \underbrace{m_2\ (\lambda y \to k\ x)}_{k\ x} \bigvee \underbrace{m_2\ (\lambda y \to k\ y)}_{m_2\ k}) \\
& \quad (\ m_2 \text{ is a probability distribution, hence, } \textstyle\int dm_2 = 1 \text{ and } \eta\text{-reduction}\ ) \\
= \;& \lambda k \to m_1\ (\lambda x \to k\ x \bigvee m_2\ k) \\
= \;& \lambda k \to m_1\ ((\lambda x \to k\ x) \bigvee (\lambda x \to m_2\ k))
\end{aligned}$$

$$= \lambda k \to \underbrace{m_1\ (\lambda x \to k\ x)}_{m_1\ k} \bigvee \underbrace{m_1\ (\lambda x \to m_2\ k)}_{m_2\ k}$$

(η-reduction and m_1 is a probability distribution, hence, $\int dm_2 = 1$)

$$= \lambda k \to m_1\ k \bigvee m_2\ k$$

$$= (\lambda k \to m_1\ k) \bigvee (\lambda k \to m_2\ k)$$

□

A.3.2 Theorem 41

Proof. By induction on the structure of the typing derivation, performing case analysis on the final rule application.

In what follows, let $\Gamma \vdash_v V : S$, $\Gamma \vdash_c C : T$, and $\rho \in [\![\Gamma]\!]$, such that $\forall (x : S_x) \in \Gamma : \rho(x) \in \mathcal{F}_{S_x}$.

Case T-Var $\quad V = x_i \quad \dfrac{x_i : S_i \in \Gamma}{\Gamma \vdash_v x_i : S_i}$

By assumption, $[\![x_i]\!]\ \rho = \rho(x_i) \in \mathcal{F}_{S_i}$.

Case T-Unit $\quad V = \mathbf{unit} \quad \dfrac{}{\Gamma \vdash_v \mathbf{unit} : \mathbf{1}}$

By assumption, $[\![\mathbf{unit}]\!]\ \rho = \ \in \{\star\} = [\![\mathbf{1}]\!] = \mathcal{F}_{\mathbf{1}}$.

Case T-Header $\quad V = h_i \quad \dfrac{}{\Gamma \vdash_v h_i : \mathcal{H}}$

Immediately: $[\![h_i]\!]\ \rho = h_i \in \{h_i, \ldots, h_k\} = [\![\mathcal{H}]\!] = \mathcal{F}_{\mathcal{H}}$.

Case T-Lit $\quad V = i \quad \dfrac{}{\Gamma \vdash_v i : \mathbb{N}}$

Immediately: $[\![i]\!]\ \rho = i \in \mathbb{N} = [\![\mathbb{N}]\!] = \mathcal{F}_{\mathbb{N}}$.

Case T-Abs $\quad V = \lambda x{:}T_1.C \quad \dfrac{x : T_1, \Gamma \vdash_c C : T_2}{\Gamma \vdash_v \lambda x{:}T_1.C : T_1 \to T_2}$

$$\begin{aligned}
[\![\lambda x{:}T_1.C]\!]\ \rho &= \lambda v \to [\![C]\!]\ [x \mapsto v]\rho && (\text{ By definition. }) \\
&&& (\text{ By induction, } \forall v : v \in \mathcal{F}_v(T_1) \Rightarrow [\![C]\!]\ [x \mapsto v]\rho\ \mathcal{F}_c(T_2). \ ) \\
&\in \mathcal{F}_v(T_1) \to \mathcal{F}_c(T_2) \\
&= \mathcal{F}_v(T_1 \to T_2)
\end{aligned}$$

Case Atomic Computations $C = \langle P \rangle$ or $V_1 \leftarrow V_2$ or **dup**

$$\frac{\vdots}{\Gamma \vdash_c C{:}\mathbf{1}}$$

By Lemma 58: $[\![C]\!]\ \rho\ A = \mathit{return}_D\ (\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})$, then:

$$\int g\ d([\![C]\!]\ \rho\ A)$$

$$(\text{ Lemma 58 })$$

$$= \int g\ d(\mathit{return}\ (\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})) \qquad (\text{ } \textstyle\int g\ d(\mathit{return}\ x) = g(x) \text{ })$$

$$= g(\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})$$

$$(\text{ Since } \mathit{dom}(\lambda\star \to \{f_c\ \rho\ x \mid x \in A\}) = \{*\},\ \chi_{F(\mathbf{1})}(\lambda\star \to \{f_C\ \rho\ x \mid x \in A\}) = 1 \ )$$

$$= \chi_{F(\mathbf{1})}(\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})g((\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})$$

$$(\text{ } \textstyle\int g\ d(\mathit{return}\ x) = g(x) \text{ })$$

$$= \int \chi_{F(\mathbf{1})}\ g\ d(\mathit{return}\ (\lambda\star \to \{f_C\ \rho\ x \mid x \in A\}))$$

$$(\text{ Lemma 58 })$$

$$= \int \chi_{F(\mathbf{1})}\ g\ d([\![C]\!]\ \rho\ A)$$

Hence, $[\![C]\!]\ \rho\ \in \mathcal{F}_{\mathbf{1}}$.

Case T-SEQ $\qquad C = C_1;\ C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C : T_2}$

$$\int g\ d([\![C_1;\ C_2]\!]\ \rho\ A)$$

(by definition)

$$= \int g\ d\,([\![C_1]\!]\ \rho\ A) \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ \rho\ B \mid (x, B) \in \mu\})$$

($\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$)

$$= \int_\mu \int g\ d\,(\Xi\{[\![C_2]\!]\ \rho\ B \mid (x, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A)$$

(by induction for $\Gamma \vdash_c C_2 : T_2$)

(Lemma 59)

$$= \int_\mu \int \chi_{F(T_2)}\ g\ d\,(\Xi\{[\![C_2]\!]\ \rho\ B \mid (x, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A)$$

(by definition)

$$= \int_\mu \int \chi_{F(T_2)}\ g\ d([\![C_1;\ C_2]\!]\ \rho\ A)$$

Hence $[\![C_1;\ C_2]\!]\ \rho\ \in \mathcal{F}_{T_2}$.

Case T-PAR $\quad C = C_1 \,\&\, C_2 \quad$ $\dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 :}$

$$\int g \; d([\![C_1 \,\&\, C_2]\!] \; \rho \; A)$$

(by definition)

$$= \int g \; d([\![C_1]\!] \; \rho \; A \gg\!=_{\mathrm{D}} \lambda\mu_1 \to [\![C_2]\!] \; \rho \; A \gg\!=_{\mathrm{D}} \lambda\mu_2 \to return_D \; (\mu_1 \bigvee \mu_2))$$

($\int g \; d(m \gg\!= h) = \int_x \int g \; d\,(h(x)) \; dm$)

$$= \int_{\mu_1} \int_{\mu_2} \int g \; d(return_D \; (\mu_1 \bigvee \mu_2)) \; d([\![C_2]\!] \; \rho \; A) \; d([\![C_1]\!] \; \rho \; A)$$

($\int g \; d(return \; x) = g(x)$)

$$= \int_{\mu_1} \int_{\mu_2} g(\mu_1 \bigvee \mu_2) \; d([\![C_2]\!] \; \rho \; A) \; d([\![C_1]\!] \; \rho \; A)$$

(by induction)

$$= \int_{\mu_1} \chi_{F(T)}(\mu_1) \int_{\mu_2} \chi_{F(T)}(\mu_2) \; g(\mu_1 \bigvee \mu_2) \; d([\![C_2]\!] \; \rho \; A) \; d([\![C_1]\!] \; \rho \; A)$$

(distributivity)

$$= \int_{\mu_1} \int_{\mu_2} \chi_{F(T)}(\mu_1) \; \chi_{F(T)}(\mu_2) \; g(\mu_1 \bigvee \mu_2) \; d([\![C_2]\!] \; \rho \; A) \; d([\![C_1]\!] \; \rho \; A)$$

($dom(\mu \bigvee \mu_{n+1}) = dom(\mu_1) \cup dom(\mu_2),$)

(then, $\mu_1 \bigvee \mu_2 \in F(T) \iff \mu, \mu_2 \in F(T)$)

(and therefore, $\chi_{F(T)}(\mu_1)\chi_{F(T)}(\mu_2) = \chi_{F(T)}(\mu_1 \bigvee \mu_2).$)

$$= \int_{\mu_1} \int_{\mu_2} \chi_{F(T)}(\mu_1 \bigvee \mu_2)\ g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A)$$

$$(\ \textstyle\int g\ d(return\ x) = g(x)\ )$$

$$= \int_{\mu_1} \int_{\mu_2} \int \chi_{F(T)}(\mu_1 \bigvee \mu_2)\ g\ d(return\ (\mu_1 \bigvee \mu_2))\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A)$$

$$(\ \textstyle\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm\ )$$

$$= \int \chi_{F(T)}(\mu_1 \bigvee \mu_2)\ g\ d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A \gg\!= \lambda\mu_2 \to return\ (\mu_1 \bigvee \mu_2))$$

$$(\ \text{by definition}\ )$$

$$= \int \chi_{F(T)}(\mu_1 \bigvee \mu_2)\ g\ d([\![C_1 \,\&\, C_2]\!]\ \rho\ A$$

Hence, $[\![C_1 \,\&\, C_2]\!]\ \rho\ \in \mathcal{F}_{T_2}$.

Case T-CHOICE $\quad C = C_1 \oplus_r C_2 \quad$ $\dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \oplus_r C_2 :}$

$$\int g \, d([\![C_1 \oplus_r C_2]\!] \, \rho \, A)$$

(by definition)

$$= \int g \, d(r[\![C_1]\!] \, \rho \, A + (1-r)[\![C_2]\!] \, \rho \, A)$$

(distributivity of addition, scalar multiplication)

$$= r \int g \, d([\![C_1]\!] \, \rho \, A) + (1-r) \int g \, d([\![C_2]\!] \, \rho \, A))$$

(by induction)

$$= r \int \chi_{F(T)} \, g \, d([\![C_1]\!] \, \rho \, A) + (1-r) \int \chi_{F(T)} \, g \, d([\![C_2]\!] \, \rho \, A))$$

(distributivity of addition, scalar multiplication)

$$= \int \chi_{F(T)} \, g \, d(r[\![C_1]\!] \, \rho \, A + (1-r)[\![C_2]\!] \, \rho \, A)$$

(by definition)

$$= \int \chi_{F(T)} \, g \, d(r[\![C_1 \oplus_r C_2]\!] \, \rho \, A)$$

Hence $[\![C_1 \oplus rC_2]\!] \, \rho \, A \in \mathcal{F}_T$.

Case T-ITER $\quad C = {C_1}^* \quad$ $\dfrac{\Gamma \vdash_c C_1 : \mathbf{1}}{\Gamma \vdash_c {C_1}^* : \mathbf{1}}$

$$\int g \, d([\![{C_1}^*]\!] \, \rho \, A)$$

(If $\mu : [\![\mathbf{1}]\!] \rightharpoonup A$, then $\mu : \mathcal{F}_{\mathbf{1}} \rightharpoonup A$ and $dom(\mu) = \{*\}$ is finite. Hence $\mu \in F(\mathbf{1})$.)

$$= \int \chi_{F(\mathbf{1})} \, g \, d([\![{C_1}^*]\!] \, \rho \, A$$

Hence $[\![{C_1}^*]\!] \, \rho \, A \in \mathcal{F}_{\mathbf{1}}$.

Case T-PRODUCE $\quad C = \textbf{produce}\ V \quad \dfrac{\Gamma \vdash_v V : S}{\Gamma \vdash_c \textbf{produce}\ V : S}$

$$\int g\ d([\![\textbf{produce}\ V]\!]\ \rho\ A)$$

(by definition)

$$= \int g\ d(return_D\ (\lambda[\![V]\!]\ \rho \to A))$$

($\int g\ d(return\ x) = g(x)$)

$$= g(\lambda[\![V]\!]\ \rho \to A)$$

(By induction $[\![V]\!]\ \rho \in \mathcal{F}_S$, and $dom(\lambda[\![V]\!]\ \rho \to A) = \{[\![V]\!]\ \rho\}$ is finite.)

(Hence, $\lambda[\![V]\!]\ \rho \to A \in F(S)$.)

$$= \chi_{F(S)}(\lambda[\![V]\!]\ \rho \to A)\ g(\lambda[\![V]\!]\ \rho \to A)$$

($\int g\ d(return\ x) = g(x)$)

$$= \int \chi_{F(S)}\ g\ d(return\ (\lambda[\![V]\!]\ \rho \to A))$$

(by definition)

$$= \int \chi_S\ g\ d([\![\textbf{produce}\ V]\!]\ \rho\ A)$$

Hence, $[\![\textbf{produce}\ V]\!]\ \rho\ \in \mathcal{F}_c(S)$.

Case T-APP $\quad C = V_1\ V_2 \quad \dfrac{\Gamma \vdash_v V_1 : S \to T \qquad \Gamma \vdash_v V_2 : S}{\Gamma \vdash_v V_1\ V_2 : T}$

$$[\![V_1\ V_2]\!]\ \rho\ A$$

$$= [\![V_1]\!]\ ([\![V_2]\!]\ \rho)\ A$$

(By induction: $[\![V_1]\!]\ \rho \in \mathcal{F}_v(S \to T) = \mathcal{F}_v(S) \to \mathcal{F}_c(T)$ and $[\![V_2]\!]\ \rho \in \mathcal{F}_v(S)$)

$$\in \mathcal{F}_c(T)$$

Case T-To $\quad C = C_1 \textbf{ to } x.C_2 \quad \dfrac{\Gamma \vdash_c C_1 : S \qquad x : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \textbf{ to } x.C_2 : T}$

$$\int g\ d([\![C_1 \textbf{ to } x.C_2]\!]\ \rho\ A)$$

(by definition)

$$= \int g\ d([\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (x, B) \in \mu\})$$

($\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$)

$$= \int_\mu \int g\ d(\Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A)$$

(by induction)

$$= \int_\mu \chi_{F(S)} \int g\ d(\Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A)$$

($\mu \in F(S) \Rightarrow v \in \mathcal{F}_S$, then by induction, $[\![C_2]\!]\ [x \mapsto v]\rho\ \in \mathcal{F}_T$.)

(Lemma 59)

$$= \int_\mu \int \chi_{F(T)}\ g\ d(\Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (x, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A)$$

($\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$)

$$= \int \chi_{F(T)}\ g\ d([\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\})$$

(by definition)

$$= \int \chi_{F(T)}\ g\ d([\![C_1 \textbf{ to } x.C_2]\!]\ \rho\ A)$$

Hence, $[\![C_1 \textbf{ to } x.C_2]\!]\ \rho\ \in \mathcal{F}_T$. □

A.3.3 Lemma 59

Lemma 59. *Let* $M = \{m_1, \ldots, m_n\} \subseteq [\![T]\!]$, *such that* $\int f \ dm_i = \int \chi_{F(T)} \ f \ dm_i$, *for* $i = 1, \ldots, n$; *then*

$$\int f \ d(\Xi M) = \int \chi_{F(T)} \ f d(\Xi M)$$

Proof. By induction on n:

- $n = 0$

$$\int g \ d(\Xi\emptyset)$$

(by definition, $\bigvee \emptyset = \lambda x \to \bot$)

$$= \int g \ d(return \ (\lambda x \to \bot))$$

($\int g \ d(return \ x) = g(x)$)

$$= g(\lambda x \to \bot)$$

($dom(()\lambda x \to \bot)$ is finite)

$$= \chi_{F(T)}(\lambda x \to \bot) g(\lambda x \to \bot)$$

($\int g \ d(return \ x) = g(x)$)

$$= \int \chi_{F(T)} \ g \ d(return \ (\lambda x \to \bot))$$

(by definition, $\bigvee \emptyset = \lambda x \to \bot$)

$$= \int \chi_{F(T)} \ g \ d(\Xi\emptyset)$$

- Induction

$$\int g\ d(\Xi\{m_1,\ldots,m_n,m_{n+1}\})$$

(by definition)

$$= \int g\ d(\Xi\{m_1,\ldots,m_n\} \gg\!= \lambda\mu \to m_{n+1} \gg\!= \lambda\mu_{n+1} \to \mathit{return}\ (\mu \bigvee \mu_{n+1}))$$

($\int g\ d(m \gg\!= h) = \int_x \int g\ d\,(h(x))\ dm$)

$$= \int_\mu \int_{\mu_{n+1}} \int g\ d(\mathit{return}\ (\mu \bigvee \mu_{n+1}))\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

($\int g\ d(\mathit{return}\ x) = g(x)$)

$$= \int_\mu \int_{\mu_{n+1}} g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

(by induction and by assumption for m_{n+1})

$$= \int_\mu \chi_{F(T)}(\mu) \int_{\mu_{n+1}} \chi_{F(T)}(\mu_{n+1})\ g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

(distributivity)

$$= \int_\mu \int_{\mu_{n+1}} \chi_{F(T)}(\mu)\ \chi_{F(T)}(\mu_{n+1})\ g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

($\mathit{dom}(\mu \bigvee \mu_{n+1}) = \mathit{dom}(\mu) \cup \mathit{dom}(\mu_{n+1}),$)

(then, $\mu \bigvee \mu_{n+1} \in F(T) \iff \mu, \mu_{n+1} \in F(T)$)

(and therefore, $\chi_{F(T)}(\mu)\chi_{F(T)}(\mu_{n+1}) = \chi_{F(T)}(\mu \bigvee \mu_{n+1})$.)

$$= \int_\mu \int_{\mu_{n+1}} \chi_{F(T)}(\mu \bigvee \mu_{n+1})\ g(\mu \bigvee \mu_{n+1})\ dm_{n+1}\ d(\Xi\{m_1,\ldots,m_n\})$$

$$= \int \chi_{F(T)}\ g\ d(\Xi\{m_1,\ldots,m_n,m_{n+1}\}$$

□

A.3.4 Theorem 44

Proof. By induction on the structure of the typing derivation, performing case analysis on the final rule application.

In what follows, let $\Gamma \vdash_v V : S$, $\Gamma \vdash_c C : T$, and $\rho \in [\![\Gamma]\!]$, such that $\forall (x : S_x) \in \Gamma : \rho(x) \in \mathcal{C}_{S_x}$.

Case T-Var $\quad V = x_i \quad \dfrac{x_i : S_i \in \Gamma}{\Gamma \vdash_v x_i : S_i}$

By assumption, $[\![x_i]\!]\ \rho = \rho(x_i) \in \mathcal{C}_{S_i}$.

Case T-Unit $\quad V = \mathbf{unit} \quad \Gamma \vdash_v \mathbf{unit} : \mathbf{1}$

$$[\![\mathbf{unit}]\!]\ \rho = \star \in \{\star\} = [\![\mathbf{1}]\!] = \mathcal{C}_{\mathbf{1}}$$

Case T-Header $\quad V = h_i \quad \Gamma \vdash_v h_i : \mathcal{H}$

$$[\![h_i]\!]\ \rho = h_i \in \{h_1, \ldots, h_n\} \in [\![\mathcal{H}]\!] = \mathcal{C}_{\mathcal{H}}$$

Case T-Lit $\quad V = n \quad \Gamma \vdash_v n : \mathbb{N}$

$$[\![n]\!]\ \rho = n \in \mathbb{N} = [\![\mathbb{N}]\!] = \mathcal{C}_{\mathbb{N}}$$

Case T-Abs $\quad V = \lambda x : T_1 . C \quad \dfrac{x : T_1, \Gamma \vdash_c C : T_2}{\Gamma \vdash_c \lambda x : T_1 . C : T_1 \to T_2}$

$$\begin{aligned}
[\![\lambda x : T_1 . C]\!]\ \rho A &= \lambda v \to \mathcal{C}[\![[x \mapsto v]\rho]\!] && (\text{ By definition. }) \\
&(\text{ By induction, } v \in \mathcal{C}_v(T_1) \Rightarrow [\![C]\!]\ [x \mapsto v]\rho\ \in \mathcal{C}_c(T_2)\ ) \\
&\in \mathcal{C}_v(T_1) \to \mathcal{C}_c(T_2) \\
&= \mathcal{C}_v(T_1 \to T_2)
\end{aligned}$$

Case Atomic Computations $\quad C = \langle P \rangle$ or $V_1 \leftarrow V_2$ or **dup**

$$\frac{\vdots}{\Gamma \vdash_c C : \mathbf{1}}$$

By Lemma 58, we know that $[\![C]\!]\ \rho\ A = \mathit{return}_D\ (\lambda\star \to \{f_C\ \rho\ x \mid x \in A\})$, then:

1. We show that $[\![C]\!]\ \rho$ is continuous:

$$\begin{aligned}
&\bigvee_{i\geq 0} [\![C]\!]\ \rho\ A_i \\
=\ &\bigvee_{i\geq 0} (\mathit{return}\ (\lambda\star \to \{f_c(x) \mid x \in A_i\})) \\
=\ &\mathit{return}\ \left(\bigvee_{i\geq 0} (\lambda\star \to \{f_c(x) \mid x \in A_i\})\right) \\
=\ &\mathit{return}\ \left(\lambda\star \to \bigcup_{i\geq 0} \{f_c(x) \mid x \in A_i\}\right) \\
=\ &\mathit{return}\ \left(\lambda\star \to \{f_c(x) \mid x \in \bigcup_{i\geq 0} A_i\}\right) \\
=\ &[\![C]\!]\ \rho \left(\bigcup_{i\geq 0} A_i\right)
\end{aligned}$$

2. Only continuous values are produced.

$$\int g\ d\,([\![C]\!]\ \rho\ A)$$

(by definition)

$$= \int g\ d\,(\mathit{return}\ (\lambda\star \to \{f_C(x) \mid x \in A\}))$$

($\int g\ d(\mathit{return}\ x) = g(x)$)

$$= g(\lambda\star \to \{f_C(x) \mid x \in A\})$$

($\mathit{dom}(\lambda\star \to \{f_C(x) \mid x \in A\}) = \mathcal{C}_v(\mathbf{1})$)

(Thus, $\lambda\star \to \{f_C(x) \mid x \in A\} \in C(\mathbf{1})$)

$$= \chi_{C(\mathbf{1})}(\lambda\star \to \{f_C(x) \mid x \in A\})\ g(\lambda\star \to \{f_C(x) \mid x \in A\})$$

$$\left(\int g\ d(\mathit{return}\ x) = g(x) \right)$$

$$= \int \chi_{C(\mathbf{1})} g\ d\,(\mathit{return}\ (\lambda\star \to \{f_C(x) \mid x \in A\}))$$

$$\left(\text{by definition} \right)$$

$$= \int \chi_{C(\mathbf{1})} g\ d\,([\![C]\!]\ \rho\ A)$$

Thus $[\![C]\!]\ \rho\ \in \mathcal{C}_c(\mathbf{1})$.

Case T-SEQ $\qquad C = C_1;\ C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C : T_2}$

1. We show that $[\![C_1;\ C_2]\!]\ \rho\ A$ is continuous:

$$\bigvee_{i \geq 0} [\![C_1;\ C_2]\!]\ \rho\ A_i$$

$$= \bigvee_{i \geq 0} ([\![C_1]\!]\ \rho\ A_i \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ \rho\ B \mid (v, B) \in \mu\})$$

$$\left(\text{continuity of } \gg\!=_{\mathrm{D}} \right)$$

$$= \bigvee_{i \geq 0} ([\![C_1]\!]\ \rho\ A_i) \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ \rho\ B \mid (v, B) \in \mu\}$$

$$\left(\text{By induction, } [\![C_1]\!]\ \rho \text{ is continuous} \right)$$

$$= [\![C_1]\!]\ \rho\ \left(\bigcup_{i \geq 0} A_i \right) \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ \rho\ B \mid (v, B) \in \mu\}$$

$$= [\![C_1;\ C_2]\!]\ \rho\ \left(\bigcup_{i \geq 0} A_i \right)$$

2. And only produces continuous values

$$
\begin{aligned}
& \int g \, d([\![C_1;\, C_2]\!]\ \rho\ A) \\
= \quad & \int g \, d([\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ \rho\ B \mid (v, B) \in \mu\} \\
= \quad & \int_{\mu} \int g \, d(\Xi\{[\![C_2]\!]\ \rho\ B \mid (v, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A) \\
& \qquad\qquad (\text{ Continuity, by induction on } \Gamma \vdash_c C_2 : T_2. \ ) \\
= \quad & \int_{\mu} \int g \, d([\![C_2]\!]\ \rho \bigcup_{(x,B)\in\mu} B)\ d([\![C_1]\!]\ \rho\ A \\
& \qquad\qquad (\text{ By induction on } \Gamma \vdash_c C_2 : T_2. \ ) \\
= \quad & \int_{\mu} \int \chi_{\mathcal{C}_{T_2}} g \, d([\![C_2]\!]\ \rho \bigcup_{(x,B)\in\mu} B)\ d([\![C_1]\!]\ \rho\ A \\
& \qquad\qquad (\text{ Continuity }) \\
= \quad & \int_{\mu} \int \chi_{\mathcal{C}_{T_2}} g \, d(\Xi\{[\![C_2]\!]\ \rho\ B \mid (v, B) \in \mu\})\ d([\![C_1]\!]\ \rho\ A \\
= \quad & \int \chi_{\mathcal{C}_{T_2}} g; d([\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to [\![C_2]\!]\ \rho \bigcup_{(x,B)\in\mu} B) \\
= \quad & \int \chi_{\mathcal{C}_{T_2}} g \, d([\![C_1;\, C_2]\!]\ \rho\ A)
\end{aligned}
$$

Thus $[\![C_1;\, C_2]\!]\ \rho \in \mathcal{C}_{T_2}$.

Case T-PAR $\qquad C = C_1 \,\&\, C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \,\&\, C_2 :}$

1. We show that $[\![C_1 \,\&\, C_2]\!]\ \rho$ is continuous:

$$\begin{aligned}
&\bigvee_{i\geq 0} [\![C_1 \,\&\, C_2]\!]\ \rho\ A_i \\
=\ &\bigvee_{i\geq 0} \Big([\![C_1]\!]\ \rho\ A_i \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A_i \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \Big) \\
&\qquad\qquad (\ \text{Continuity of } \gg\!=_{\mathrm{D}}\ ) \\
=\ &\bigvee_{i\geq 0} ([\![C_1]\!]\ \rho\ A_i) \gg\!= \lambda\mu_1 \to \bigvee_{i\geq 0} ([\![C_2]\!]\ \rho\ A_i) \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \\
&\qquad\qquad (\ [\![C_1]\!]\ \rho \text{ and } [\![C_2]\!]\ \rho \text{ are continuous by induction.}\ ) \\
=\ &[\![C_1]\!]\ \rho\ (\bigcup_{i\geq 0} A_i) \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i) \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \\
=\ &[\![C_1 \,\&\, C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)
\end{aligned}$$

2. Only continuous values are produced:

$$\begin{aligned}
&\int g\ d([\![C_1 \,\&\, C_2]\!]\ \rho\ A) \\
=\ &\int g\ d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2) \\
=\ &\int_{\mu_1} \int_{\mu_2} g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
=\ &\int_{\mu_1} \chi_{C(T)}(\mu_1) \int_{\mu_2} \chi_{C(T)}(\mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
&\qquad\qquad (\ \text{Induction on } \Gamma \vdash_c C_1 : T, \Gamma \vdash_c C_2 : T\ ) \\
=\ &\int_{\mu_1} \int_{\mu_2} \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A) \\
&\qquad\qquad (\ \mu_1, \mu_2 \in C(T) \iff \mu_1 \bigvee \mu_2 \in C(T)\ ) \\
&\qquad\qquad (\ \text{Hence } \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2) = \chi_{C(T)}(\mu_1)\chi_{C(T)}(\mu_2)\ ) \\
=\ &\int_{\mu_1} \int_{\mu_2} \chi_{C(T)}(\mu_1 \bigvee \mu_2) g(\mu_1 \bigvee \mu_2)\ d([\![C_2]\!]\ \rho\ A)\ d([\![C_1]\!]\ \rho\ A)
\end{aligned}$$

$$= \int \chi_{C(T)} g \; d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to [\![C_2]\!]\ \rho\ A \gg\!= \lambda\mu_2 \to \mathit{return}\ (\mu_1 \bigvee \mu_2))$$

$$= \int \chi_{C(T)} g \; d([\![C_1 \,\&\, C_2]\!]\ \rho\ A)$$

Thus $[\![C_1 \,\&\, C_2]\!]\ \rho \in \mathcal{C}_T$.

Case T-CHOICE $\qquad C = C_1 \oplus_r C_2 \qquad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1 \oplus_r C_2 :}$

1. We show that $[\![C_1 \oplus C_2]\!]\ \rho$ is continuous.

$$\bigvee_{i\geq 0} [\![C_1 \oplus C_2]\!]\ \rho\ A_i$$

$$= \bigvee_{i\geq 0} (r([\![C_1]\!]\ \rho\ A_i) + (1-r)([\![C_2]\!]\ \rho\ A_i))$$

$$= r \bigvee_{i\geq 0} ([\![C_1]\!]\ \rho\ A_i) + (1-r) \bigvee_{i\geq 0} ([\![C_2]\!]\ \rho\ A_i))$$

(by induction)

$$= r([\![C_1]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)) + (1-r)([\![C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i))$$

$$= [\![C_1 \oplus C_2]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)$$

2. Only continous values are produced:

$$\int g \; d([\![C_1 \oplus_r C_2]\!]\ \rho\ A)$$

$$= r \int g \; d([\![C_1]\!]\ \rho\ A) + (1-r) \int g \; d([\![C_2]\!]\ \rho\ A)$$

(by induction)

$$= r \int \chi_{C(T)} g \; d([\![C_1]\!]\ \rho\ A) + (1-r) \int \chi_{C(T)} g \; d([\![C_2]\!]\ \rho\ A)$$

$$= \int \chi_{C(T)} g \, d\big(r([\![C_1]\!]\ \rho\ A) + (1-r)([\![C_2]\!]\ \rho\ A)\big)$$

$$= \int \chi_{C(T)} g \, d\big([\![C_1 \oplus_r C_2]\!]\ \rho\ A)$$

Thus, $[\![C_1 \oplus C_2]\!]\ \rho \in \mathcal{C}_T$.

Case T-ITER $\qquad C = {C_1}^* \qquad \dfrac{\Gamma \vdash_c C_1 : \mathbf{1}}{\Gamma \vdash_c {C_1}^* : \mathbf{1}}$

1. We show that $[\![{C_1}^*]\!]\ \rho$ is continuous:

$$\bigvee_{i\geq 0} ([\![{C_1}^*]\!]\ \rho\ A_i)$$

$$= \bigvee_{i\geq 0} (\bigvee_{n\geq 0} [\![C_1^n]\!]\ \rho\ A_i)$$

$$= \bigvee_{n\geq 0} [\![C_1^n]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)$$

$$= [\![{C_1}^*]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)$$

2. Only continuous values are produced:

$$\int g \, d([\![{C_1}^*]\!]\ \rho\ A)$$

$$= \int g \, d(\bigvee_{n\geq 0} [\![C_1^n]\!]\ \rho\ A)$$

($\mathcal{C}_{\mathbf{1}} = [\![\mathbf{1}]\!] = \{*\}$, then clearly $\mu \in [\![\mathbf{1}]\!] \rightharpoonup 2^{PH} = C(\mathbf{1})$)

$$= \int \chi_{C(\mathbf{1})} g \, d(\bigvee_{n\geq 0} [\![C_1^n]\!]\ \rho\ A)$$

$$= \int \chi_{C(\mathbf{1})} g \, d([\![{C_1}^*]\!]\ \rho\ A)$$

Thus $[\![C]\!]\ \rho\ \in \mathcal{C}_{\mathbf{1}}$.

Case T-PRODUCE $\qquad C = \mathbf{produce}\ V \qquad \dfrac{\Gamma \vdash_v V : S}{\Gamma \vdash_c \mathbf{produce}\ V : S}$

1. We show that $[\![\mathbf{produce}\ V]\!]\ \rho$ is continuous.

$$\begin{aligned}
& \bigvee_{i\geq 0} [\![\mathbf{produce}\ V]\!]\ \rho\ A_i \\
= & \bigvee_{i\geq 0} \mathit{return}\ (\lambda [\![V]\!]\ \rho \to A_i) \\
= & \ \mathit{return} \bigvee_{i\geq 0} (\lambda [\![V]\!]\ \rho \to A_i) \\
= & \ \mathit{return}\ (\lambda [\![V]\!]\ \rho \to \bigcup_{i\geq 0} A_i) \\
= & \ [\![\mathbf{produce}\ V]\!]\ \rho\ (\bigcup_{i\geq 0} A_i)
\end{aligned}$$

2. Only continuous values are produced:

$$\begin{aligned}
& \int g\ d([\![\mathbf{produce}\ V]\!]\ \rho\ A \\
= & \int g\ d(\mathit{return}\ (\lambda [\![V]\!]\ \rho \to A)) \\
= & \ g(\lambda [\![V]\!]\ \rho \to A) \\
& \qquad\qquad (\ \Gamma \vdash_c V : S, \text{ by induction } V \in \mathcal{C}_S\ ) \\
= & \ \chi_{C(S)}(\lambda [\![V]\!]\ \rho \to A) g(\lambda [\![V]\!]\ \rho \to A) \\
= & \int \chi_{C(S)} g\ d(\mathit{return}\ (\lambda [\![V]\!]\ \rho \to A)) \\
= & \int \chi_{C(S)} g\ d([\![\mathbf{produce}\ V]\!]\ \rho\ A)
\end{aligned}$$

Thus $[\![\mathbf{produce}\ V]\!]\ \rho \in \mathcal{C}_S$.

Case T-App $\quad C = V_1\ V_2 \quad$ $\dfrac{\Gamma \vdash_v V_1 : S \to T \qquad \Gamma \vdash_v V_2 : S}{\Gamma \vdash_c V_1\ V_2 : T}$

$$[\![V_1\ V_2]\!]\ \rho$$

(by definition)

$$= \ [\![V_1]\!]\ \rho\ ([\![V_1]\!]\ \rho)$$

(by induction: $[\![V_1]\!]\ \rho \in \mathcal{C}_v(S \to T) = \mathcal{C}_v(S) \to \mathcal{C}_c(T)$ and $[\![V_2]\!]\ \rho \in \mathcal{C}_v(S)$)

$$\in \ \mathcal{C}_c(T)$$

Case T-To $\quad C = C_1\ \mathbf{to}\ x.C_2 \quad$ $\dfrac{\Gamma \vdash_c C_1 : S \qquad x : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1\ \mathbf{to}\ x.C_2 : T}$

1. We show that $[\![C_1\ \mathbf{to}\ x.C_2]\!]\ \rho$ is continuous:

$$\bigvee_{i \geq 0} ([\![C_1\ \mathbf{to}\ x.C_2]\!]\ \rho\ A_i)$$

$$= \ \bigvee_{i \geq 0} ([\![C_1]\!]\ \rho\ A_i \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\})$$

(continuity of $\gg\!=_{\mathrm{D}}$)

$$= \ \bigvee_{i \geq 0} ([\![C_1]\!]\ \rho\ A_i) \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}$$

(by induction)

$$= \ [\![C_1]\!]\ \rho\ \Big(\bigcup_{i \geq 0} A_i\Big) \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}$$

2. All produced values are continuous:

$$\int g\ d([\![C_1\ \mathbf{to}\ x.C_2]\!]\ \rho\ A)$$

$$= \ \int g\ d([\![C_1]\!]\ \rho\ A \gg\!= \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\})$$

$$= \int_\mu \int g \; d(\Xi\{[\![C_2]\!] \; [x \mapsto v]\rho \; B \mid (v, B) \in \mu\}) \; d([\![C_1]\!] \; \rho \; A)$$

(by induction $[\![C_2]\!] \; [x \mapsto v]\rho \in \mathcal{C}_T$)

(then $\int g \; d([\![C_2]\!] \; [x \mapsto v]\rho \; A) = \int \chi_{C(T)} g \; d([\![C_2]\!] \; [x \mapsto v]\rho \; A)$)

(Lemma 60)

$$= \int_\mu \int \chi_{C(T)} g \; d(\Xi\{[\![C_2]\!] \; [x \mapsto v]\rho \; B \mid (v, B) \in \mu\}) \; d([\![C_1]\!] \; \rho \; A)$$

$$= \int_\mu \int \chi_{C(T)} g \; d(\Xi\{[\![C_2]\!] \; [x \mapsto v]\rho \; B \mid (v, B) \in \mu\}) \; d([\![C_1]\!] \; \rho \; A)$$

$$= \int \chi_{C(T)} \; g \; d([\![C_1 \; \mathbf{to} \; x.C_2]\!] \; \rho \; A$$

Thus $[\![C_1 \; \mathbf{to} \; x.C_2]\!] \; \rho \in \mathcal{C}_T$. □

A.3.5 Lemma 60

Lemma 60. *Let $M = \{m_1, \ldots, m_n\} \subseteq [\![T]\!]$, such that if*

$$\int f \; dm_i = \int \chi_{C(T)} \; f \; dm_i,$$

for all $i = 1, \ldots, n$; then

$$\int f \; d(\Xi M) = \int \chi_{C(T)} \; f d(\Xi M)$$

Proof.

$$
\begin{aligned}
& \Xi M \\
= & \int_{\mu_1} \cdots \int_{\mu_n} \int g \; d(return \; (\mu_1 \bigvee \cdots \mu_n)) \; dm_n \cdots dm_1 \\
= & \int_{\mu_1} \chi_{C(T)}(\mu_1) \cdots \int_{\mu_n} \chi_{C(T)}(\mu_n) \int g \; d(return \; (\bigvee_{i=1}^{n} \mu_i)) \; dm_n \cdots dm_1 \\
& \qquad (\text{ If } \mu_1, \ldots, \mu_n \in C(T), \text{ then } \mu_1 \bigvee \cdots \bigvee \mu_n \in C(T). \;) \\
& \qquad (\text{ Since } dom(\mu_1 \bigvee \cdots \bigvee \mu_n) = dom(\mu_1) \cup \cdots \cup dom(\mu_n) \subseteq \mathcal{C}_T \;) \\
= & \int_{\mu_1} \chi_{C(T)}(\mu_1) \cdots \int_{\mu_n} \chi_{C(T)}(\mu_n) \int \chi_{C(T)} \; g \; d(return \; (\bigvee_{i=1}^{n} \mu_i)) \; dm_n \cdots dm_1 \\
= & \int_{\mu_1} \cdots \int_{\mu_n} \int \chi_{C(T)} \; g \; d(return \; (\bigvee_{i=1}^{n} \mu_i)) \; dm_n \cdots dm_1
\end{aligned}
$$

□

A.3.6 Theorem 45

Recall the definition of PNK semantics given in Appendix A.2. To prove Theorem 45, we actually prove the following lemmas:

Lemma 61. *Let P be a closed predicate. Then for all $A \in 2^{PH}$:*

$$return_D \; (\llbracket P \rrbracket^p \; * \; A) = \llbracket P \rrbracket_{PNK} \; A$$

Proof. By induction on the structure of predicates:

Cases drop,skip, tests $f = n$: straightforward by definition.

Case $\neg P$

$$\begin{aligned} \mathit{return}_D\ (\llbracket \neg P \rrbracket^p\ *\ A) &= \mathit{return}_D\ (A - \llbracket P \rrbracket^p\ *\ A) \\ &= \mathit{return}_D\ (\llbracket P \rrbracket^p\ *\ A) \gg\!=_{\mathrm{D}} \lambda B \to \mathit{return}\ (A - B) \\ &= \llbracket P \rrbracket_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda B \to \mathit{return}\ (A - B) \\ &= \llbracket \neg P \rrbracket_{PNK}\ A \end{aligned}$$

Case $P_1 \wedge P_2$

$$\begin{aligned} & \mathit{return}_D\ (\llbracket P_1 \wedge P_2 \rrbracket^p\ *\ A) \\ = \ & \mathit{return}_D\ \{h \in A \mid B_{P_1 \wedge P_2}\ *\ h\} \\ = \ & \mathit{return}_D\ \{h \in A \mid B_{P_1}\ *\ h \text{ and } B_{P_2}\ *\ h\} \\ = \ & \mathit{return}_D\ \{h \in \{h \in A \mid B_{P_1}\ *\ h\} \mid B_{P_2}\ *\ h\} \\ = \ & \mathit{return}_D\ \{h \in A \mid B_{P_1}\ *\ h\} \gg\!=_{\mathrm{D}} \lambda A' \to \mathit{return}_D\ \{h \in A' \mid B_{P_2}\ *\ h\} \\ = \ & \llbracket P_1 \rrbracket_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda A' \to \llbracket P_2 \rrbracket_{PNK}\ A' \\ = \ & \llbracket P_1 \rrbracket_{PNK}\ A \gg\!=_{\mathrm{D}} \llbracket P_2 \rrbracket_{PNK} \end{aligned}$$

Case $P_1 \vee P_2$

$$\begin{aligned} & \mathit{return}_D\ (\llbracket P_1 \vee P_2 \rrbracket^p\ *\ A) \\ = \ & \mathit{return}_D\ (\llbracket P_1 \rrbracket^p\ *\ A \cup \llbracket P_2 \rrbracket^p\ *\ A) \\ = \ & \mathit{return}_D\ (\llbracket P_1 \rrbracket^p\ *\ A) \gg\!=_{\mathrm{D}} \lambda B \to \mathit{return}_D\ (\llbracket P_2 \rrbracket^p\ *\ A) \gg\!=_{\mathrm{D}} \lambda C \to \\ & \mathit{return}_D\ (B \cup C) \\ = \ & \llbracket P_1 \rrbracket_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda B \to \llbracket P_2 \rrbracket_{PNK}\ A \gg\!=_{\mathrm{D}} \lambda C \to \mathit{return}_D\ (B \cup C) \\ = \ & \llbracket P_1 \vee P_2 \rrbracket_{PNK}\ A \end{aligned}$$

□

Lemma 62. *Let C be a closed probabilistic computation. Define*

$$\psi(m) = m \gg\!=_{\mathrm{D}} \lambda \mu \to \mathit{return}_D\ \big(\mu(*)\big)$$

then for all $A \in 2^{PH}$*:*

$$\psi([\![C]\!] \ * \ A) = [\![C]\!]_{PNK} \ A$$

Conservativity (Theorem 45) is a consequence of this lemma.

Proof. By induction on the structure of probabilistic computations. Note that we have the following computation rules for ψ:

$$\begin{aligned}
\psi(\mathit{return}_D \ (\lambda\star \to A)) &= \mathit{return}_D \ A \\
\psi(m \gg\!=_{\mathrm{D}} f) &= m \gg\!=_{\mathrm{D}} (\psi \circ f) \\
\psi(r \cdot m) &= r \cdot \psi(m) \\
\psi(m_1 + m_2) &= \psi(m_1) + \psi(m_2) \\
\psi(\bigvee_{i \geq 0} m_i) &= \bigvee_{i \geq 0} \psi(m_i)
\end{aligned}$$

Atomic Computations For atomic computations C, we have

$$\begin{aligned}
\psi([\![C]\!] \ * \ A) &= \psi(\mathit{return}_D \ (\lambda\star \to \{f_C \ * \ h \mid h \in A\})) \\
&= \mathit{return}_D \ \{f_C \ * \ h \mid h \in A\} \\
&\quad \text{(Immediate for } f \leftarrow n \text{ and } \mathbf{dup}\text{, for predicates this follows from Lemma 61)} \\
&= [\![C]\!]_{PNK} \ A
\end{aligned}$$

Sequential composition

$$
\begin{aligned}
\psi\left(\llbracket C_1;\, C_2 \rrbracket \;*\; A\right) &= \psi\left(\llbracket C_1 \rrbracket \;*\; A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{\llbracket C_2 \rrbracket \;*\; B \mid (v, B) \in \mu\}\right) \\
&= \psi\left(\llbracket C_1 \rrbracket \;*\; A \gg\!=_{\mathrm{D}} \lambda\mu \to \llbracket C_2 \rrbracket \;*\; \mu(x)\right) \\
&= \llbracket C_1 \rrbracket \;*\; A \gg\!=_{\mathrm{D}} \lambda\mu \to \psi\left(\llbracket C_2 \rrbracket \;*\; \mu(x)\right) \\
&= \llbracket C_1 \rrbracket \;*\; A \gg\!=_{\mathrm{D}} \lambda\mu \to \llbracket C_2 \rrbracket_{PNK}\; \mu(*) \qquad (\text{ induction }) \\
&= \llbracket C_1 \rrbracket \;*\; A \gg\!=_{\mathrm{D}} \lambda\mu \to \mathit{return}_D\; \mu(*) \gg\!=_{\mathrm{D}} \llbracket C_2 \rrbracket_{PNK} \\
&= \psi\left(\llbracket C_1 \rrbracket \;*\; A\right) \gg\!=_{\mathrm{D}} \llbracket C_2 \rrbracket_{PNK} \\
&= \psi(\llbracket C_1 \rrbracket \;*\; A) \gg\!=_{\mathrm{D}} \llbracket C_2 \rrbracket_{PNK} \\
&= \llbracket C_1 \rrbracket_{PNK}\; A \gg\!=_{\mathrm{D}} \llbracket C_2 \rrbracket_{PNK} \qquad (\text{ induction })
\end{aligned}
$$

Parallel composition

$$
\begin{aligned}
& \psi\left(\llbracket C_1 \,\&\, C_2 \rrbracket \;*\; A\right) \\
=\;& \psi\left(\llbracket C_1 \rrbracket \;*\; A \gg\!= \lambda\mu_1 \to \llbracket C_2 \rrbracket \;*\; A \gg\!= \lambda\mu_2 \to \mathit{return}_D\; (\mu_1 \bigvee \mu_2)\right) \\
=\;& \llbracket C_1 \rrbracket \;*\; A \gg\!= \lambda\mu_1 \to \llbracket C_2 \rrbracket \;*\; A \gg\!= \lambda\mu_2 \to \mathit{return}_D\; (\mu_1(*) \cup \mu_2(*)) \\
=\;& \llbracket C_1 \rrbracket \;*\; A \gg\!= \lambda\mu_1 \to \mathit{return}_D\; \mu_1(*) \gg\!= \lambda A_1 \to \\
& \llbracket C_2 \rrbracket \;*\; A \gg\!= \lambda\mu_2 \to \mathit{return}_D\; \mu_2(*) \gg\!= \lambda A_2 \to \mathit{return}_D\; (A_1 \cup A_2) \\
=\;& \psi\left(\llbracket C_1 \rrbracket \;*\; A\right) \gg\!= \lambda A_1 \to \psi\left(\llbracket C_2 \rrbracket \;*\; A\right) \gg\!= \lambda A_2 \to \mathit{return}_D\; (A_1 \cup A_2) \\
=\;& \llbracket C_1 \rrbracket_{PNK}\; A \gg\!= \lambda A_1 \to \llbracket C_2 \rrbracket_{PNK}\; A \gg\!= \lambda A_2 \to \mathit{return}_D\; (A_1 \cup A_2) \\
=\;& \llbracket C_1 \,\&\, C_2 \rrbracket_{PNK}\; * A
\end{aligned}
$$

Probabilistic Choice

$$\begin{aligned}
& \psi([\![C_1 \oplus_r C_2]\!] \ * \ A) \\
= \ & \psi\left(r[\![C_1]\!] \ * \ A + (1-r)([\![C_2]\!] \ * \ A)\right) \\
= \ & r \cdot \psi\left([\![C_1]\!] \ * \ A\right) + (1-r) \cdot \psi\left([\![C_2]\!] \ * \ A\right) \\
= \ & r([\![C_1]\!]_{PNK} \ A) + (1-r)([\![C_2]\!]_{PNK} \ A) \\
= \ & [\![C_1 \oplus_r C_2]\!]_{PNK} \ A)
\end{aligned}$$

Iteration

$$\begin{aligned}
& \psi([\![C^*]\!] \ * \ A) \\
= \ & \psi(\bigvee_{n \geq 0} ([\![C^n]\!] \ * \ A) \\
= \ & \bigvee_{n \geq 0} \psi([\![C^n]\!] \ * \ A) \\
= \ & \bigvee_{n \geq 0} [\![C^n]\!]_{PNK} \ A \\
= \ & [\![C^*]\!]_{PNK} \ A
\end{aligned}$$

□

A.3.7 Theorem 47

Proof. We prove the following slightly stronger statement: Let R_1, R_2 be terminals such that $\Vdash_c R_1 : T$, $\Vdash_c R_2 : \mathbf{1}$ and $R_1 \rightsquigarrow R_2$, then $[\![R_1]\!] = [\![R_2]\!]$ if $T = \mathbf{1}$, otherwise $[\![R_2]\!] = [\![R_1; \ \mathbf{skip}]\!]$.

We proceed by induction on the elaboration relation. In the first three cases of $\rightsquigarrow$, the result is immediate.

In what follows, assume $R_{11} \rightsquigarrow R_{21}$ and $R_{12} \rightsquigarrow R_{22}$, $\Vdash_c R_{11} : T_1$, $\Vdash_c R_{11} : T_2$,

If $R_{11};\ R_{12} \rightsquigarrow R_{21};\ R_{22}$, then $[\![R_{21};\ R_{22}]\!] = [\![R_{11};\ R_{22}]\!]$, by induction independently of T_1. Then either $[\![R_{21};\ R_{22}]\!] = [\![R_{11};\ R_{12}]\!]$ or $[\![R_{21};\ R_{22}]\!] = [\![R_{11};\ R_{12};\ \mathbf{skip}]\!]$ according to T_2.

If $R_{11} \,\&\, R_{12} \rightsquigarrow R_{21} \,\&\, R_{22}$, then the required follows immediately by induction, inversion and the definition of the denotational semantics.

If $R_{11} \oplus R_{12} \rightsquigarrow R_{21} \oplus R_{22}$, then Note that $T_1 = T_2$, the required result then either follows immediately, by induction, or by induction and distributivity of ; over $\oplus$.

Assume $R_1 \rightsquigarrow R_2$. It is easy to show by induction on n that if $[\![R_1]\!] = [\![R_2]\!]$, then $[\![R_1^n]\!] = [\![R_2^n]\!]$ for all n. It then follows that $[\![{R_1}^*]\!] = [\![{R_2}^*]\!]$

The penultimate case follows by inversion and the definition of the denotational semantics. The final case is immediate after inversion. □

A.3.8 Theorem 49

Proof. The proof proceeds by induction on the structure of the evaluation rules and case analysis on the final rule.

In cases E-Pred, E-Mod, E-Dup, E-Prod and C is a terminal. By reflection, $C = R$, Hence $[\![C]\!] = [\![R]\!]$.

In cases E-Seq, E-Par, E-Choice, the required follows by unfolding the definition and applying the induction hypothesis.

Case E-Iter Note that

$$[\![C^*]\!]\ \rho\ A = \bigvee_{n\geq 0} [\![C^n]\!]\ \rho\ A \text{ and} [\![R^*]\!]\ \rho\ A = \bigvee_{n\geq 0} [\![R^n]\!]\ \rho\ A$$

Then we we need to show that $\forall n \geq 0 : [\![C^n]\!] = [\![R^n]\!]$. This follows by induction on n.

Case E-AppAbs

$$\begin{aligned} [\![\lambda x{:}T.C\ V]\!]\ \rho\ A &= [\![\lambda x{:}T.C]\!]\ \rho\ [\![V]\!]\ \rho\ A \\ &= [\![C]\!]\ [x \mapsto [\![V]\!]\ ]\rho\ A \\ &= [\![[x \mapsto V]C]\!]\ \rho\ A \qquad (\text{ Lemma 63 }) \end{aligned}$$

Case E-ToProduce

$$
\begin{aligned}
& [\![C_1 \textbf{ to } x.C_2]\!]\ \rho\ A \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition)} \\
=\ & [\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by induction)} \\
=\ & [\![\textbf{produce } V]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition)} \\
=\ & \mathit{return}_D\ (\lambda[\![V]\!]\ \rho \to A) \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(left-unit of } \gg\!=_{\mathrm{D}} \text{)} \\
=\ & \Xi\{[\![C_2]\!]\ [x \mapsto ([\![V]\!]\ \rho)]\rho\ A\} \\
=\ & [\![C_2]\!]\ [x \mapsto ([\![V]\!]\ \rho)]\rho\ A \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Lemma 63)} \\
=\ & [\![[x \mapsto V]C_2]\!]\ \rho\ A \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by induction)} \\
=\ & [\![R_2]\!]\ \rho\ A
\end{aligned}
$$

Case E-ToPred / E-ToMod / E-ToDup All atomic computations follow the same scheme, i.e. assume C_1 is a computation that evaluates to R, where R is atomic.

$$
\begin{aligned}
& [\![C_1 \textbf{ to } x.C_2]\!]\ \rho\ A \\
=\ & [\![C_1]\!]\ \rho\ A\lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by induction)} \\
=\ & [\![R]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}
\end{aligned}
$$

$$= \quad [\![R]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}$$

($dom(\mu) = \{*\}$)

$$= \quad [\![R]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto *]\rho\ B \mid (v, B) \in \mu\}$$

$$= \quad [\![R]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto [\![\mathbf{unit}]\!]\ \rho\ ]\rho\ B \mid (v, B) \in \mu\}$$

(Lemma 63)

$$= \quad [\![R]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![[x \mapsto \mathbf{unit}]C_2]\!]\ \rho\ B \mid (v, B) \in \mu\}$$

(by induction)

$$= \quad [\![R]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![R_2]\!]\ \rho\ B \mid (v, B) \in \mu\}$$

$$= \quad [\![R_1;\ R_2]\!]\ \rho\ A$$

Case E-TolIter

$$
\begin{aligned}
& [\![C_1 \ \mathbf{to}\ x.C_2]\!]\ \rho\ A \\
= \ & [\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ \mu(v) \mid v \in dom(\mu)\} \\
= \ & [\![{R_1}^*]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ \mu(v) \mid v \in dom(\mu)\}
\end{aligned}
$$

(by inversion)

$$
= \ [\![{R_1}^*]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to [\![C_2]\!]\ [x \mapsto *]\rho\ \mu(*)
$$

(Lemma 63)

$$
= \ [\![{R_1}^*]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to [\![[x \mapsto \mathbf{unit}]C_2]\!]\ \rho\ \mu(*)
$$

$$
= \ [\![{R_1}^*]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to [\![[x \mapsto \mathbf{unit}]C_2]\!]\ \rho\ \Big(\bigcup_{x \in dom(\mu)} \mu(x)\Big)
$$

(by induction)

$$
= \ [\![{R_1}^*]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to [\![R_2]\!]\ \rho\ \Big(\bigcup_{x \in dom(\mu)} \mu(x)\Big)
$$

(by definition)

$$
= \ [\![{R_1}^*;\ R_2]\!]\ \rho\ A
$$

Case E-ToSeq

$$
\begin{aligned}
& [\![C_1 \ \mathbf{to}\ x.C_2]\!]\ \rho\ A \\
= \ & [\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{ by induction }) \\
= \ & [\![R_{11};\ R_{12}]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \\
& \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\} \\
= \ & [\![R_{11}]\!]\ \rho\ A\lambda\mu_1 \to \\
& (\Xi\{[\![R_{12}]\!]\ \rho\ B \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v']\rho\ B' \mid (v', B') \in \mu\} \mid (v, B) \in \mu_1\}) \\
= \ & [\![R_{11}]\!]\ \rho\ A\lambda\mu_1 \to \Xi\{[\![R_2]\!]\ \rho\ A \mid (v, B) \in \mu_1\} \\
= \ & [\![R_{11};\ R_2]\!]\ \rho\ A
\end{aligned}
$$

Case E-ToChoice

$$
\begin{aligned}
& [\![C_1 \ \mathbf{to}\ x.C_2]\!]\ \rho\ A \\
= \ & [\![C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{ by induction }) \\
= \ & [\![R_{11} \oplus_r R_{12}]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\} \\
= \ & (r[\![R_{11}]\!]\ \rho\ A + (1-r)[\![R_{12}]\!]\ \rho\ A) \gg\!=_{\mathrm{D}} \lambda\mu \to \\
& \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\} \\
= \ & (r[\![R_{11}]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}) \\
& + (1-r)([\![R_{12}]\!]\ \rho\ A\Xi\{[\![C_2]\!]\ [x \mapsto v]\rho\ B \mid (v, B) \in \mu\}) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{ by induction }) \\
= \ & r([\![R_{21}]\!]\ \rho\ A) + (1-r)([\![R_{22}]\!]\ \rho\ A) \\
= \ & [\![R_{21} \oplus r R_{22}]\!]\ \rho\ A
\end{aligned}
$$

Case E-ToPar Note that the inversion for $\Gamma \Vdash_c C_1 \,\&\, C_2 : T$ says that $T = \mathbf{1}$. Then the argument proceeds analogously to E-ToIter. □

A.3.9 Lemma 63

Lemma 63. *Let $V, \widehat{V}$ be values and C be a computation such that $x : \widehat{S}, \Gamma \vdash_v V : S$, $\Gamma \vdash_v \widehat{V} : \widehat{S}$ and $x : \widehat{S}, \Gamma \vdash_c C : T$. Let $\rho \in [\![\Gamma]\!]$ then*

$$[\![V]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho \quad = \quad [\![[x \mapsto \widehat{V}]V]\!]\ \rho$$

$$[\![C]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho \quad = \quad [\![[x \mapsto \widehat{V}]C]\!]\ \rho$$

Proof. By induction on the structure of the typing derivation. We can ignore T-Unit, T-Header, T-Lit, T-Skip,T-Drop,T-Dup, since they don't contain any variables.

Case Atomic Computations $C = P$ or $V_1 \leftarrow V_2$ or **dup**

$$\frac{\vdots}{\Gamma \vdash_c C : \mathbf{1}}$$

For predicates we can prove:

- Negation

$$\begin{aligned} B_{\neg P}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho &= \neg B_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \\ &= \neg B_{[x \mapsto \widehat{V}]P}\ \rho \\ &= B_{\neg[x \mapsto \widehat{V}]P}\ \rho \\ &= B_{[x \mapsto \widehat{V}]\neg P}\ \rho \end{aligned}$$

- Disjunction

$$\begin{aligned} B_{P_1 \vee P_2}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho &= B_{P_1}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \text{ or } B_{P_2}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \\ &= B_{[x \mapsto \widehat{V}]P_1}\ \rho \text{ or } B_{[x \mapsto \widehat{V}]P_2}\ \rho \\ &= B_{[x \mapsto \widehat{V}(P_1 \vee P_2)}\ \rho \end{aligned}$$

  And similar for T-Conj

- Guard

$$\begin{aligned} B_{V_1=V_2}\ ([x \mapsto [\![\widehat{V}]\!]\ ]\rho)\ (\pi :: h) &= \pi.[\![V_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho = [\![V_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho \\ &= \pi.[\![[x \mapsto \widehat{V}]V_1]\!]\ \rho[\![[x \mapsto \widehat{V}]V_2]\!]\ \rho \\ &= B_{[x\mapsto\widehat{V}]V_1=V_2}\ \rho\ (pi :: h) \end{aligned}$$

Hence, $B_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho = B_{[x\mapsto\widehat{V}]P}\ \rho$, and

$$\begin{aligned} f_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ h &= \begin{cases} h & \text{if} B_P\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ h \\ \bot & \text{otherwise} \end{cases} \\ &= \begin{cases} h & \text{if} B_{[x\mapsto\widehat{V}]P}\ \rho\ h \\ \bot & \text{otherwise} \end{cases} \\ &= f_{[x\mapsto\widehat{V}]P}\ \rho\ h \end{aligned}$$

For T-Mod:

$$\begin{aligned} f_{V_1\leftarrow V_2}\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ (\pi :: h) &= \pi[[\![V_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \mapsto [\![V_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho] :: h \\ &= \pi[[\![[x \mapsto \widehat{V}]V_1]\!]\ \rho \mapsto [\![[x \mapsto \widehat{V}]V_2]\!]\ \rho] :: h \\ &= f_{[x\mapsto\widehat{V}]V_1\leftarrow V_2}\ \rho\ (\pi :: h) \end{aligned}$$

Thus we can conclude that for an atomic computation C,

$$\begin{aligned} [\![C]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A &= \mathit{return}_D\ (\lambda\star \to \{f_C\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ h \mid h \in A\}) \\ &= \mathit{return}_D\ (\lambda\star \to \{f_{[x\mapsto\widehat{V}]C}\ \rho\ h \mid h \in A\}) \\ &= [\![[x \mapsto \widehat{V}]C]\!]\ \rho\ A \end{aligned}$$

Case T-SEQ $\quad C = C_1;\ C_2 \quad \dfrac{\Gamma \vdash_c C_1 : T_1 \qquad \Gamma \vdash_c C_2 : T_2}{\Gamma \vdash_c C : T_2}$

$$
\begin{aligned}
& [\![C_1;\ C_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
= \ & [\![C_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![C_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ B \mid (v, B) \in \mu\} \\
& \qquad\qquad (\text{ induction }) \\
= \ & [\![[x \mapsto \widehat{V}]C_1]\!]\ \rho\ A \gg\!=_{\mathrm{D}} \lambda\mu \to \Xi\{[\![[x \mapsto \widehat{V}]C_2]\!]\ \rho\ B \mid (v, B) \in \mu\} \\
= \ & [\![[x \mapsto \widehat{V}]C_1;\ C_2]\!]\ \rho\ A
\end{aligned}
$$

Case T-PAR $\quad C = C_1 \,\&\, C_2 \quad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C : T}$

$$
\begin{aligned}
& [\![C_1;\ C_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
= \ & [\![C_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \gg\!=_{\mathrm{D}} \lambda\mu_1 \to \\
& [\![C_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \gg\!=_{\mathrm{D}} \lambda\mu_2 \to \\
& \mathit{return}_D\ (\mu_1 \bigvee \mu_2) \\
& \qquad\qquad (\text{ by induction }) \\
= \ & [\![[x \mapsto \widehat{V}]C_1]\!]\ \rho\ A \gg\!= \lambda\mu_1 \to \\
& [\![[x \mapsto \widehat{V}]C_2]\!]\ \rho\ A \gg\!= \lambda\mu_2 \to \\
& \mathit{return}_D\ (\mu_1 \bigvee \mu_2) \\
= \ & [\![(C_1 \,\&\, C_2)[x \mapsto \widehat{V}]]\!]\ \rho\ A
\end{aligned}
$$

Case T-CHOICE $\quad C = C_1 \oplus C_2 \quad \dfrac{\Gamma \vdash_c C_1 : T \qquad \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C : T}$

$$
\begin{aligned}
& [\![C_1 \oplus C_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
= \ & r[\![C_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A + (1-r)([\![C_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{ by induction }) \\
= \ & r[\![[x \mapsto \widehat{V}]C_1]\!]\ \rho\ A + (1-r)([\![[x \mapsto \widehat{V}]C_2]\!]\ \rho\ A) \\
= \ & [\![[x \mapsto \widehat{V}]C_1 \oplus C_2]\!]\ \rho\ A
\end{aligned}
$$

Case T-PRODUCE $\quad C = \mathbf{produce}\ V \quad \dfrac{\Gamma \vdash_v V : S}{\Gamma \vdash_c \mathbf{produce}\ V : T}$

$$
\begin{aligned}
& [\![\mathbf{produce}\ V]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
= \ & \mathit{return}_D\ (\lambda [\![V]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \to A) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{ by induction }) \\
= \ & \mathit{return}_D\ (\lambda [\![[x \mapsto \widehat{V}]V]\!]\ \rho A \to) \\
= \ & [\![[x \mapsto \widehat{V}]\mathbf{produce}\ V]\!]\ \rho\ A
\end{aligned}
$$

Case T-ABS $\quad V = \lambda x : S.C \quad \dfrac{x : S, \Gamma \vdash_c C : T}{\Gamma \vdash_v \lambda X : S.C : S \to T}$

- Assume $x \neq y$

$$
\begin{aligned}
[\![\lambda y : S.C]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho &= \lambda v \to [\![C]\!]\ [y \mapsto v, x \mapsto [\![\widehat{V}]\!]\ \rho]\rho \\
&= \lambda v \to [\![[x \mapsto \widehat{V}]C]\!]\ [y \mapsto v]\rho \quad (\text{ induction }) \\
&= [\![[x \mapsto \widehat{V}]\lambda y : S.C]\!]\ \rho
\end{aligned}
$$

- Assume $x = y$

$$
\begin{aligned}
[\![\lambda y : S.C]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho &= \lambda v \to [\![C]\!]\ [y \mapsto v]\rho \\
&= [\![[x \mapsto \widehat{V}]\lambda y : S.C]\!]\ \rho
\end{aligned}
$$

Case T-ITER $C = {C_1}^*$ $\dfrac{\Gamma \vdash_c C_1 : \mathbf{1}}{\Gamma \vdash_c {C_1}^* \mathbf{1} :}$

$$
\begin{aligned}
& [\![{C_1}^*]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho\ A \\
= & \bigvee_{n \geq 0} [\![{C_1}^n]\!]\ [x \mapsto [\![\widehat{V}]\!]\ ]\rho\ A \\
= & \bigvee_{n \geq 0} [\![[x \mapsto \widehat{V}]{C_1}^n]\!]\ \rho\ A \\
= & \ [\![[x \mapsto \widehat{V}]({C_1}^*)]\!]\ \rho\ A
\end{aligned}
$$

Case T-APP $C = V_1\ V_2$ $\dfrac{\Gamma \vdash_v V_1 : S \to T \qquad \Gamma \vdash_v V_2 : S}{\Gamma \vdash_c V\ V : T}$

$$
\begin{aligned}
& [\![V_1\ V_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
= & \ [\![V_1]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\ [\![V_2]\!]\ [x \mapsto [\![\widehat{V}]\!]\ \rho]\rho\ A \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{ induction }) \\
= & \ [\![[x \mapsto \widehat{V}]V_1]\!]\ \rho\ [\![[x \mapsto \widehat{V}]V_2]\!]\ \rho\ A \\
= & \ [\![[x \mapsto \widehat{V}]V_1\ [x \mapsto \widehat{V}]V_2]\!]\ \rho\ A \\
= & \ [\![[x \mapsto \widehat{V}]V_1\ V_2]\!]\ \rho\ A
\end{aligned}
$$

Case T-TO $C = C_1\ \mathbf{to}\ y.C_2$ $\dfrac{\Gamma \vdash_c C_1 : S \qquad y : S, \Gamma \vdash_c C_2 : T}{\Gamma \vdash_c C_1\ \mathbf{to}\ y.C_2 : T}$

- Assume $x \neq y$:

$$\begin{aligned}
& [\![C_1 \textbf{ to } y.C_2]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \\
= \ & [\![C_1]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \gg\!= \lambda\mu \rightarrow \\
& \Xi\{[\![C_2]\!]\ [y \mapsto v, x \mapsto [\![V]\!]\ \rho]\rho\ \mu(v) \mid v \in dom(\mu)\} \\
= \ & [\![[x \mapsto \widehat{V}]C_1]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow \Xi\{[\![[x \mapsto \widehat{V}]C_2]\!]\ [y \mapsto v]\rho\ \mu(v) \mid v \in dom(\mu)\} \\
= \ & [\![[x \mapsto \widehat{V}](C_1 \textbf{ to } y.C_2)]\!]\ \rho\ A
\end{aligned}$$

- Assume $x = y$:

$$\begin{aligned}
& [\![C_1 \textbf{ to } y.C_2]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \\
= \ & [\![C_1]\!]\ [x \mapsto [\![V]\!]\ \rho]\rho\ A \gg\!= \lambda\mu \rightarrow \\
& \Xi\{[\![C_2]\!]\ [y \mapsto v]\rho\ \mu(v) \mid v \in dom(\mu)\} \\
= \ & [\![[x \mapsto \widehat{V}]C_1]\!]\ \rho\ A \gg\!= \lambda\mu \rightarrow \Xi\{[\![C_2]\!]\ [y \mapsto v]\rho\ \mu(v) \mid v \in dom(\mu)\} \\
= \ & [\![[x \mapsto \widehat{V}](C_1 \textbf{ to } y.C_2)]\!]\ \rho\ A
\end{aligned}$$

□

Lemma 64. *Let $V, \widehat{V}$ be a values and C be a computation such that $x : \widehat{S}, \Gamma \Vdash_v V : S$, $\Gamma \Vdash_v \widehat{V} : \widehat{S}$ and $x : \widehat{S}, \Gamma \Vdash_c C : T$. Let $\rho[\![\Gamma]\!]$ then*

$$\begin{aligned}
[\![V]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho &= [\![[x \mapsto \widehat{V}]V]\!]\ \rho \\
[\![C]\!]\ [x \mapsto ([\![\widehat{V}]\!]\ \rho)]\rho &= [\![[x \mapsto \widehat{V}]C]\!]\ \rho
\end{aligned}$$

Proof. Since $\Gamma \Vdash_v V : S$ implies $\Gamma \vdash_v V : S$ and $\Gamma \Vdash_c C : T$ implies $\Gamma \vdash_c C : T$ the required follows from Lemma 63. □

Index

$D(X)$, 95
$M(X)$, 97, 98
$\gg\!=_{\mathrm{M}}$, 99
$\mathcal{C}_c(T)$, 104
$\mathcal{C}_v(T)$, 104
$\rightsquigarrow$, 108
$\mathcal{F}_c(T)$, 103
$\mathcal{F}_v(T)$, 103
$\|_D$, 99
$\uparrow$, 49
$\mathbf{C}[T]$, 111
$\mathbf{T}[T]$, 111
$\mathbf{V}[T]$, 111
$\rightharpoonup$, 97
$\Downarrow$, 109
$return_M$, 99
Ξ, 99

ALS-Prolog, 29
Anglican, 90

Borel set, 93
B-Prolog, 29

CBN, 83
CBV, 83
    Fine-Grained (FG-CBV), 74, 83
`Coerce`, 64
complete lattice, 12
conservativity, 105
continuity, 104
correctness, 26, 27

denotational semantics, 48, 97
    $\mathcal{R}[\![\cdot]\!]$, 41
    $\mathcal{R}[\![\cdot]\!]_L$, 49
    $[\![\cdot]\!]$, 40, 42
    $[\![\cdot]\!]_L$, 48, 50
    P$\lambda\omega$NK
        $[\![C]\!]_{PNK}$, 105
        computations $[\![C]\!]$, 100
        predicates $[\![P]\!]^p$, 100
        types $[\![T]\!]$, 98
        values $[\![V]\!]$, 100

effect handlers, 37
    generalised, 54
elaboration, 108

Fine-Grained CBV (FG-CBV), 74, 83
finite ascending chain, 59
finite maps, 102
fixed point fusion, 26
Frenetic, 89

Gen, 90
generalised model, 25
generalised semantics, 26
gossip protocols, 112
greedy strategy, 27

Herbrand

base, 11
interpretation, 12

immediate consequence operator, 12
extended, 20
generalised, 25
Interpretation, 12
I_P, 19
iteration, 84

Kleene Algebra with Test (KAT), 90
GKAT, 90
Kleene fixed point theorem, 13, 61
Knaster–Tarski, 13

λ-abstraction, 74, 82
`Lattice`, 56
`liftRec`, 58
logical relation, 103, 104, 111

McNetKat, 90
measure, 93
monad
free, 37, 54
monotonicity, 27, 59
montone iteration, 49
mutual recursion, 61

`ND`, 37
NetKAT, 89
non-determinism, 33, 35
non-deterministic choice, 33

ω-chain, 92
ωCPO, 92
OpenFlow, 89
ωQBS, 75, 84, 94

P4, 89
packet, 79
packet history, 79
P$\lambda\omega$NK, 73
power domain, 95
Probabilistic NetKAT (PNK), 73
probability measure, 93
prototype, 77

QBS, 75, 94

reduction, 109
`runND`, 57
`runRec`, 57
runTable, 43
`runTableD`, 46
`runTableF`, 59
`runTableM`, 64

Software Defined Networks (SDN), 89
Stan, 90
stratification, 13, 23
strong normalisation, 76, 87, 111
SWI-Prolog, 29
syntax, 85

`Table`, 39
table model, 20, 22
`TableF`, 56
`TableM`, 62
thunk, 83
T_P, 12
$\widehat{T}_P$, 20
T_P^L, 25
Turing, 90
type system, 87
typing judgement
$\Gamma \Vdash_c C : T$, 107
$\Gamma \vdash_c C : T$, 88
$\Gamma \Vdash_v V : T$, 107
$\Gamma \vdash_v V : T$, 88
$\Gamma \models \mathbf{wf}\ P$, 88

U_P, 18
$U_P^{\perp}$, 19

Venture, 90

XSB, 19, 29

Yap, 18, 29

Bibliography

Aerts, B., Goedemé, T., and Vennekens, J. (2016). A probabilistic logic programming approach to automatic video montage. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, ECAI'16, pages 234–242, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Anderson, C. J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., and Walker, D. (2014). Netkat: semantic foundations for networks. In *POPL*, pages 113–126. ACM.

Apt, K. R., Blair, H. A., and Walker, A. (1988). *Towards a Theory of Declarative Knowledge.* Morgan Kaufmann.

Arias, J. and Carro, M. (2016). Description and evaluation of a generic design to integrate CLP and tabled execution. In *PPDP*, pages 10–23. ACM.

Arias, J. and Carro, M. (2019). Incremental evaluation of lattice-based aggregates in logic programming using modular TCLP. In *PADL*, volume 11372 of *Lecture Notes in Computer Science*, pages 98–114. Springer.

Arntzenius, M. and Krishnaswami, N. (2020). Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.*, 4(POPL):22:1–22:28.

Arntzenius, M. and Krishnaswami, N. R. (2016). Datafun: a functional datalog. In *ICFP*, pages 214–227. ACM.

Aumann, R. J. et al. (1961). Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630.

Bacci, G., Furber, R., Kozen, D., Mardare, R., Panangaden, P., and Scott, D. (2018). Boolean-valued semantics for the stochastic λ-calculus. In *LICS*, pages 669–678. ACM.

Backhouse, R. C. (2000). Galois connections and fixed point calculus. In Backhouse, R. C., Crole, R. L., and Gibbons, J., editors, *Algebraic*

and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, volume 2297 of *LNCS*, pages 89–148. Springer.

Battenfeld, I., Schröder, M., and Simpson, A. (2007). A convenient category of domains. *Electr. Notes Theor. Comput. Sci.*, 172:69–99.

Bauer, A. and Pretnar, M. (2015). Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123.

Bird, R. and Rabe, F. (2019). How to calculate with nondeterministic functions. In *MPC*, volume 11825 of *Lecture Notes in Computer Science*, pages 138–154. Springer.

Bird, R. S. and de Moor, O. (1997). *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, 76(1).

Ceri, S., Gottlob, G., and Tanca, L. (1989). What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., Silva, C., and Rocha, R. (2008). An improved continuation call-based implementation of tabling. In *Practical Aspects of Declarative Languages, 10th International Symposium*, volume 4902 of *LNCS*, pages 197–213. Springer.

Cusumano-Towner, M. F., Saad, F. A., Lew, A. K., and Mansinghka, V. K. (2019). Gen: a general-purpose probabilistic programming system with programmable inference. In *PLDI*, pages 221–236. ACM.

Dahlqvist, F. and Kozen, D. (2020). Semantics of higher-order probabilistic programs with conditioning. *Proc. ACM Program. Lang.*, 4(POPL):57:1–57:29.

Danielsson, N. A., Hughes, J., Jansson, P., and Gibbons, J. (2006). Fast and loose reasoning is morally correct. In *POPL*, pages 206–217. ACM.

Danos, V. and Ehrhard, T. (2011). Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Inf. Comput.*, 209(6):966–991.

Danvy, O. and Nielsen, L. R. (2001). Defunctionalization at work. In *PPDP*, pages 162–174. ACM.

Darais, D., Labich, N., Nguyen, P. C., and Horn, D. V. (2017). Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP):12:1–12:25.

De Maeyer, D., Renkens, J., Cloots, L., De Raedt, L., and Marchal, K. (2013). Phenetic: network-based interpretation of unstructured gene lists in e. coli. *Molecular BioSystems*, 9(7):1594–1603.

De Raedt, L., Kimmig, A., and Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, pages 2462–2467.

de Vries, E. and Löh, A. (2014). True sums of products. In *WGP@ICFP*, pages 83–94. ACM.

Desouter, B., van Dooren, M., and Schrijvers, T. (2015). Tabling as a library with delimited control. *Theory Pract. Log. Program.*, 15(4-5):419–433.

Devriese, D. and Piessens, F. (2011). Explicitly recursive grammar combinators - A better model for shallow parser dsls. In *PADL*, volume 6539 of *Lecture Notes in Computer Science*, pages 84–98. Springer.

Devriese, D. and Piessens, F. (2012). Finally tagless observable recursion for an abstract grammar model. *J. Funct. Program.*, 22(6):757–796.

Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J., and Madhavapeddy, A. (2015). Effective concurrency through algebraic effects. In *OCaml Workshop*, page 13.

Ehrhard, T. and Tasson, C. (2019). Probabilistic call by push value. *Logical Methods in Computer Science*, 15(1).

Fierens, D., den Broeck, G. V., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted boolean formulas. *TPLP*, 15(3):358–401.

Foster, N., Guha, A., Reitblatt, M., Story, A., Freedman, M. J., Katta, N. P., Monsanto, C., Reich, J., Rexford, J., Schlesinger, C., et al. (2013). Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134.

Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., and Silva, A. (2016). Probabilistic NetKAT. In Thiemann, P., editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 282–309. Springer.

Gacek, A. (2008). The abella interactive theorem prover (system description). In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer.

Ge, H., Xu, K., and Ghahramani, Z. (2018). Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690.

Goodman, N. D. (2013). The principles and practice of probabilistic programming. In *POPL*, pages 399–402. ACM.

Goodman, N. D., Mansinghka, V. K., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. (2012). Church: a language for generative models. *CoRR*, abs/1206.3255.

Goubault-Larrecq, J. (2019). A probabilistic and non-deterministic call-by-push-value language. In *Thirty-Fourth Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, Canada, June 24–27, 2019, Proceedings*, page To appear.

Goubault-Larrecq, J. and Varacca, D. (2011). Continuous random variables. In *LICS*, pages 97–106. IEEE Computer Society.

Guo, H.-F. and Gupta, G. (2004). Simplifying dynamic programming via tabling. In *Practical Aspects of Declarative Languages*, volume 3057 of *LNCS*, pages 163–177. Springer.

Guo, H.-F. and Gupta, G. (2008). Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience*, 38(1):75–94.

Heunen, C., Kammar, O., Staton, S., and Yang, H. (2017). A convenient category for higher-order probability theory. In *LICS*, pages 1–12. IEEE Computer Society.

Hinze, R. (2000). Deriving backtracking monad transformers. In *ICFP*, pages 186–197. ACM.

Jeannin, J., Kozen, D., and Silva, A. (2013). Language constructs for non-well-founded computation. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 61–80. Springer.

Jeuring, J. and Swierstra, S. D. (1995). Constructing functional programs for grammar analysis problems. In *FPCA*, pages 259–269. ACM.

Jones, C. and Plotkin, G. D. (1989). A probabilistic powerdomain of evaluations. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 186–195. IEEE.

Jones, S. L. P., Vytiniotis, D., Weirich, S., and Washburn, G. (2006). Simple unification-based type inference for gadts. In *ICFP*, pages 50–61. ACM.

Jung, A. and Tix, R. (1998). The troublesome probabilistic powerdomain. *Electr. Notes Theor. Comput. Sci.*, 13:70–91.

Kisa, D., den Broeck, G. V., Choi, A., and Darwiche, A. (2014). Probabilistic sentential decision diagrams. In *KR*. AAAI Press.

Kiselyov, O. and Ishii, H. (2015). Freer monads, more extensible effects. In *Haskell*, pages 94–105. ACM.

Kiselyov, O., Sabry, A., and Swords, C. (2013). Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM.

Kiselyov, O., Shan, C., Friedman, D. P., and Sabry, A. (2005). Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP*, pages 192–203. ACM.

Kowalski, R. A. (1974). Predicate logic as programming language. In *IFIP Congress*, pages 569–574. North-Holland.

Kozen, D. (1981). Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350.

Kozen, D. (1997). Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443.

Kozen, D. and Smith, F. (1996). Kleene algebra with tests: Completeness and decidability. In *CSL*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259. Springer.

Levy, P. B. (2001). *Call-by-push-value*. PhD thesis, Queen Mary University of London, UK.

Lloyd, J. W. (1984). *Foundations of Logic Programming*. Springer-Verlag, New York.

Lu, A. (2016). Venture: An extensible platform for probabilistic metaprogramming. Master's thesis, Massachusetts Institute of Technology.

MacNeille, H. M. (1937). Partially ordered sets. *Transactions of the American Mathematical Society*, pages 416–460.

McBride, C. (2012). *The Frank Manual*.

McBride, C. (2015). Turing-completeness totally free. In *MPC*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer.

McBride, C. and Paterson, R. (2008). Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13.

Moggi, E. (1991). Notions of computation and monads. *Inf. Comput.*, 93(1):55–92.

Mokhov, A., Lukyanov, G., Marlow, S., and Dimino, J. (2019). Selective applicative functors. *Proc. ACM Program. Lang.*, 3(ICFP):90:1–90:29.

Oliveira, B. C. and Cook, W. R. (2012). Functional programming with structured graphs. In *ICFP*, pages 77–88. ACM.

Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Swift, T., and Warren, D. S. (1997). *Computer Aided Verification: 9th International Conference, Haifa, Israel, June 22–25, 1997 Proceedings*, pages 143–154. Springer.

Reich, J., Monsanto, C., Foster, N., Rexford, J., and Walker, D. (2013). Modular SDN programming with Pyretic. *Technical Report of USENIX*.

Reynolds, J. C. (1998). Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397.

Riguzzi, F., Wielemaker, J., and Zese, R. (2018). Probabilistic inference in swi-prolog. In *PLP@ILP*, volume 2219 of *CEUR Workshop Proceedings*, pages 15–27. CEUR-WS.org.

Saheb-Djahromi, N. (1980). Cpo's of measures for nondeterminism. *Theoretical Computer Science*, 12(1):19–37.

Santos, J. and Rocha, R. (2013). On the efficient implementation of mode-directed tabling. In *Practical Aspects of Declarative Languages*, volume 7752 of *LNCS*, pages 141–156. Springer.

Santos Costa, V., Rocha, R., and Damas, L. (2012). The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34.

Schrijvers, T., Costa, V. S., Wielemaker, J., and Demoen, B. (2008). Towards typed prolog. In *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 693–697. Springer.

Schrijvers, T., Demoen, B., Desouter, B., and Wielemaker, J. (2013). Delimited continuations for Prolog. *Theory Pract. Log. Program.*, 13(4-5):533–546.

Schrijvers, T., Piróg, M., Wu, N., and Jaskelioff, M. (2019). Monad transformers and modular algebraic effects: what binds them together. In *Haskell@ICFP*, pages 98–113. ACM.

Scibior, A., Kammar, O., and Ghahramani, Z. (2018). Functional programming for modular bayesian inference. *PACMPL*, 2(ICFP):83:1–83:29.

Scott, D. and Strachey, C. (1971). *Toward a mathematical semantics for computer languages*, volume 1.

Smolka, S., Eliopoulos, S. A., Foster, N., and Guha, A. (2015). A fast compiler for NetKAT. In *ICFP*, pages 328–341. ACM.

Smolka, S., Foster, N., Hsu, J., Kappé, T., Kozen, D., and Silva, A. (2020). Guarded kleene algebra with tests: verification of uninterpreted programs in nearly linear time. *Proc. ACM Program. Lang.*, 4(POPL):61:1–61:28.

Smolka, S., Kahn, D. M., Kumar, P., Foster, N., Kozen, D., and Silva, A. (2017a). Deciding probabilistic program equivalence in NetKAT. *CoRR*, abs/1707.02772.

Smolka, S., Kumar, P., Foster, N., Kozen, D., and Silva, A. (2017b). Cantor meets Scott: semantic foundations for probabilistic networks. In *POPL*, pages 557–571. ACM.

Smolka, S., Kumar, P., Kahn, D. M., Foster, N., Hsu, J., Kozen, D., and Silva, A. (2019). Scalable verification of probabilistic networks. In *PLDI*, pages 190–203. ACM.

Staton, S. (2017). Commutative semantics for probabilistic programming. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 855–879. Springer.

Swadi, K. N., Taha, W., Kiselyov, O., and Pasalic, E. (2006). A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, pages 160–169. ACM.

Swierstra, W. (2008). Data types à la carte. *J. Funct. Program.*, 18(4):423–436.

Swift, T. (1999). Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240.

Swift, T. and Warren, D. (2010). Tabling with answer subsumption: Implementation, applications performance. pages 300–312.

Swift, T. and Warren, D. S. (2012). XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187.

Tait, W. W. (1967). Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212.

Vákár, M., Kammar, O., and Staton, S. (2019). A domain theory for statistical probabilistic programming. *PACMPL*, 3(POPL):36:1–36:29.

Van Hentenryck, P., Degimbe, O., Charlier, B. L., and Michel, L. (1993). Abstract interpretation of Prolog based on OLDT resolution. Technical report, Providence, RI, USA.

Wadler, P. (1985). How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer.

Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). SWI-Prolog. *Theory Pract. Log. Program.*, 12(1-2):67–96.

Wood, F. D., van de Meent, J., and Mansinghka, V. (2014). A new approach to probabilistic programming inference. In *AISTATS*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 1024–1032. JMLR.org.

Wu, N. and Schrijvers, T. (2015). Fusion for free - efficient algebraic effect handlers. In *MPC*, volume 9129 of *Lecture Notes in Computer Science*, pages 302–322. Springer.

Zhou, N.-F. (2012). The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218.

Zhou, N. F. and Dovier, A. (2011). A tabled Prolog program for solving sokoban. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 896–897.

Zhou, N.-F., Kameya, Y., and Sato, T. (2010). Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *22nd International Conference on Tools with Artificial Intelligence (ICTAI), 2010*, volume 2, pages 213–218.

List of Publications

Journal Articles

Alexander Vandenbroucke, Maciej Piróg, Benoit Desouter, and Tom Schrijvers. 2016. Tabling with Sound Answer Subsumption. In Theory and Practice of Logic Programming. Cambridge University Press, 2016, Vol. 16, iss. 5, pp. 933–949. DOI: http://doi.org/10.1017/S147106841600048X

Alexander Vandenbroucke and Tom Schrijvers. 2019. P$\lambda\omega$NK: functional probabilistic NetKAT. Proc. ACM Program. Lang. 4, POPL, Article 39 (January 2020), 27 pages. DOI: https://doi.org/10.1145/3371107

Papers at International Conferences and Symposia

Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. 2015. Fixing non-determinism. In Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (IFL '15). Association for Computing Machinery, New York, NY, USA, Article 5, 1-–12. DOI: https://doi.org/10.1145/2897336.2897342

Benoit Desouter, Marko van Dooren, Tom Schrijvers, and Alexander Vandenbroucke. Tabling as a library with delimited control. IJCAI International Joint Conference on Artificial Intelligence, volume 2016–January, pages 4145–4149, AAAI Press, 2275 East Bayshore Road, Suite 160, Palo Alto CA 94303 USA, International Joint Conference on Artificial Intelligence, New York, 9/7/2016 – 15/7/2016. https://dl.acm.org/doi/abs/10.5555/3061053.3061235

Alexander Vandenbroucke. The functional perspective on advanced logic programming. Leibniz International Proceedings in Informatics, LIPIcs, volume 52, pages 20-, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany, ICLP, New York City, USA, 16/10/2016 – 21/10/2016. DOI: https://doi.org/10.4230/OASIcs.ICLP.2016.20

Papers at Workshops

Alexander Vandenbroucke, and Tom Schrijvers. 2017. From PRISM to ProbLog and back again. CEUR Workshop Proceedings, volume 1916, issue 4, pages 26-40, PLP, Orléans, France, 7/9/2017 - 7/9/2017. http://ceur-ws.org/Vol-1916/

Posters and Extended Abstracts at Workshops

Alexander Vandenbrouke and Tom Schrijvers. 2017. ProbLog and Applicative Probabilistic Programming Workshop on Probabilistic Programming Semantics (PPS), Paris, France, 17-01-2017.

Alexander Vandenbroucke and Tom Schrijvers. 2018. Towards Lambda Abstractions for Probabilistic NetKAT PROBPROG2018, MIT, Boston, MA, United States of America, 05/08/2018-07/08/2018.



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
alexander.vandenbroucke@kuleuven.be
https://dtai.cs.kuleuven.be