KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Contributions in Programming Languages Theory

Logical Relations and Type Theory

Amin Timany

Supervisors:
Prof. dr. B. Jacobs
Prof. dr. ir. F. Piessens

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

May 2018

Contributions in Programming Languages Theory

Logical Relations and Type Theory

Amin TIMANY

Examination committee:
Prof. dr. ir. P. Van Houtte, chair (private defense)
Prof. dr. ir. C. Vandecasteele, chair (public defense)
Prof. dr. B. Jacobs, supervisor
Prof. dr. ir. F. Piessens, supervisor
Prof. dr. M. Denecker
Prof. dr. ir. G. Janssens
Prof. dr. L. Birkedal
(Aarhus University, Denmark)
Dr. M. Sozeau
(Université Paris 7 Diderot, France)

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

May 2018

Preface

During the past few years, working as a PhD researcher, I learned a lot and grew both as a researcher and personally. I would like to thank my PhD advisor Bart Jacobs for his steadfast support. He was always present for discussions and to help me wherever and whenever I needed him. I learned a lot from him. I am thankful to Frank Piessens for being my co-advisor; his support and encouragement meant a lot to me. I would also like to specially thank Lars Birkedal. My short and long term visits to his research group at Aarhus university and our collaborations were of paramount importance in shaping this thesis and my career. I cannot overstate the effect of his unwavering support in shaping this thesis and my research career. I would also like to express my gratitude towards Matthieu Sozeau for his support and collaboration and Nicolas Tabareau for supporting my visit to Matthieu Sozeau at Inria Paris & Université Paris Diderot. I am grateful to all my collaborators: Andreas Abel, Lars Birkedal, Arthur Charguéraud, Jesper Cockx, Dominique Devriese, Derek Dreyer, Bart Jacobs, Jacques-Henri Jourdan, Ralf Jung, Jan-Oliver Kaiser, Robbert Krebbers, Morten Krogh-Jespersen, Willem Penninckx, Léo Stefanesco, Matthieu Sozeau, Joseph Tassarotti and Philip Wadler.

I thank my PhD examination committee: the members of my supervisory committee Bart Jacobs, Frank Piessens, Marc Denecker and Gerda Janssens, the external members Lars Birkedal and Matthieu Sozeau, and the chairman of the examination committee Paul Van Houtte (private defense). I am grateful to Carlo Vandecasteele for agreeing to be the chair of the public defense of my PhD on such short notice.

I would like to thank all the members of the DistriNet research group, the department of computer science and KU Leuven for providing a great working environment for me. I am very thankful to my friends and colleagues in Aarhus university who made my time there very enjoyable and productive. I would specially like to mention the following people: Aslan Askarov, Johan Bay, Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Kristof Coninx, Wilfried

Daniëls, Javier del Cid, Dominique Devriese, Thomas Dinsdale-Young, Jafar Hamin, Emad Heydari Beni, Mathias Høier, Kristoffer Just Andersen, Robbert Krebbers, Morten Krogh-Jespersen, Jan Tobias Mühlberg, Andreas Nuyts, Marit Ohlenbusch, Willem Penninckx, Andrew Polonsky, Malte Schwerhoff, Lau Skorstengaard, Bas Spitters, Stelios Tsampas, Alexander van den Berghe, Neline van Ginkel, Gijs Vanspauwen, Mathias Vorreiter Pedersen and Thomas Winant.

I would like to thank my dear family, my parents, my wife, my brother and my sister for their support and caring. I am particularly thankful to my wife Arezoo. Her love, care, support and patience throughout my PhD was absolutely essential.

My special gratitude goes to anonymous reviewers who reviewed my submissions; both those that resulted in acceptance and those that did not. They all provided useful constructive feedback that improved the quality of my work.

Last but not least, I am thankful for the financial support of the following projects: EU FP7 FET-Open project ADVENT under grant number 308830, Flemish Research Fund grants G.0058.13, G.0962.17N. I am grateful to the following funding opportunities for supporting my research visits: CoqHoTT ERC Grant 637339, ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU), EUTypes funding ECOST-STSM-CA15123-150816-080859, Flemish Research Fund grant V435817N and EUTypes funding ECOST-STSM-CA15123-40667.

May 24$^{\text{th}}$ 2018,
Leuven, Belgium

Abstract

Software systems are ubiquitous. Failure in safety- and security-critical systems, e.g., the control system of a passenger plane, can be quite catastrophic. Hence, it is crucial to ensure that safety- and security-critical software systems are correct. Types play an important role in helping us achieve this goal. They help compilers check for (some) programmer's mistakes. They also form the basis of a group of proof assistants. A proof assistant is a software that allows for formalization and mechanization of mathematics, including the theory of types, theory of programming languages, program verification, etc. In this thesis we contribute to the study of programming languages and type theory. In the first part of this thesis we formalize a mathematical theory (closely related to the theory of types and programming languages) in the proof assistant Coq and contribute to the type theory of this proof assistant. In the second part of this thesis we study a number of programming languages through their type systems. In particular, we develop methods for proving soundness (correctness) of their type systems and to prove equivalence of programs.

The first part of this thesis begins with the formal specification of category theory, a mathematical theory closely related to the theory of types and programming languages, in the proof assistant Coq. Coq is a proof assistant based on the predicative calculus of inductive constructions (pCIC). In this formalization we have taken advantage of a feature of Coq known as universe polymorphism to represent (relative) smallness and largeness of categories using Coq's universe hierarchy. The formalization of category theory that is presented in this part reveals a shortcoming of pCIC: the fact that the cumulativity relation of Coq, also known as the subtyping relation, needs to be extended to support subtyping of inductive types. The cumulativity relation for categories (represented using inductive types) would allow a small category to be also considered a large category, i.e., the type of small categories would be a subtype of the type of large categories. The following chapter presents the predicative calculus of cumulative inductive constructions (pCuIC). pCuIC extends pCIC to solve the aforementioned shortcoming of pCIC by introducing a novel subtyping relation

for inductive types. The cumulativity relation introduced for inductive types also has interesting consequences for types that do not involve the mathematical concept of smallness and largeness. For instance, it unifies (judgementally equates) the type *list A* of lists (whose elements are terms of type A) at all universe levels. This novel subtyping relation has been integrated into the proof assistant Coq as of the official release of Coq 8.7.

In the second part of this thesis we develop logical relation models, as operationally-based semantic approaches for proving type soundness and establishing equivalence of programs, for a number of programming languages. We develop logical relations models for $\mathsf{F}_{\mu,ref,conc}$, a concurrent ML-like programming language, and use them to prove type soundness and equivalence of programs. We use the latter to establish the equivalence of concurrent counter and stack modules. One of the main results of this thesis is developing a logical relations model which allows us to establish proper encapsulation of state in the programming language STLang. STLang is a programming language featuring a Haskell-style ST monad which is used to encapsulate state. This problem was open for almost two decades. We solve this problem by showing that certain program equivalences hold in the presence of the ST monad that would not hold if the state was not properly encapsulated. Finally, we develop logical relations for an extension of $\mathsf{F}_{\mu,ref,conc}$ with continuations, a feature that makes reasoning about programs quite intricate. We develop a reasoning principle for our system which allows us to treat parts of the program that do not involve continuations in a disrupting manner (but can nevertheless interact with other parts which do) as though there are no continuations in the programming language. We use this novel reasoning principle together with our logical relations model to prove that an implementation of a web server that uses continuations (in the style of Racket web servers) is equivalent to one which does not use continuations.

It is well known that developing and working with logical relations models for advanced type systems such as those studied in the second part of this thesis is very intricate. In this thesis we mitigate this issue by working in the Iris program logic. Iris is a state-of-the-art program logic based on separation logic for verification of higher-order concurrent imperative programs. Working in Iris allows us to develop our logical relations models at a higher level of abstraction and thus avoid the usual intricacies associated with developing and working with such models. Furthermore, we take advantage of the formalization of Iris on top of the Coq proof assistant to mechanize all of the results in this part of the thesis on top of Coq.

Beknopte samenvatting

Softwaresystemen zijn overal. Storingen in veiligheidskritische systemen zoals het controlesysteem van een vliegtuig, kunnen katastrofaal zijn. Daarom is het heel belangrijk om veiligheidskritische systemen te verifiëren. Types spelen een grote rol in dit verband. In compilers helpen ze met het checken van (sommige) fouten van de programmeur. Aan de andere kant vormen types de basis van een klasse van bewijsassistenten. Een bewijsassistent is een programma waarmee we wiskundige theorieën en stellingen, waaronder ook de theorie van programmeertalen, formeel kunnen definëren en bewijzen. In dit proefschrift dragen we bij aan de studie van de theorie van programmeertalen en de theorie van types. In het eerste deel van dit proefschrift formaliseren we een wiskundige theorie (die nauw verwant is aan de theorie van types en programmeertalen) in de bewijsassistent Coq. Bovendien leveren we bijdragen aan de typetheorie van Coq. In het tweede deel van dit proefschrift bestuderen we een aantal programmeertalen aan de hand van hun typesysteem. In het bijzonder ontwikkelen we methoden om soundness (correctheid) van typesystemen en gelijkwaardigheid van programma's te bewijzen.

Het eerste deel van dit proefschrift begint met formalisatie van categorietheorie, een wiskundige theorie die nauw verwant is aan de theorie van types en programmeertalen, in de bewijsassistent Coq. Coq is een bewijsassistent die gebaseerd is op de *Predicative Calculus of Inductive Constructions* (pCIC). In deze formalisering maken we gebruik van een van de kenmerken van Coq, genaamd universumpolymorfisme, om (relatieve) kleinheid en grootheid van categorieën te bepalen aan de hand van de universumhiërarchie. De formalisering van de theorie van categorieën van het eerste hoofdstuk onthult een gebrek van pCIC: het feit dat er geen cumulativiteitsrelatie (m.a.w. een subtyperelatie) geldt tussen inductieve types. De cumulativiteitsrelatie tussen categorieën (gedefinieerd als inductieve types) zou toelaten kleine categorieën te beschouwen als grote categorieën, d.w.z., het type van kleine categorieën zou een subtype zijn van dat van grote categorieën. Het tweede hoofdstuk breidt pCIC uit met een cumulativiteitsrelatie tussen inductieve types. De nieuwe cumulativiteitsrelatie

is niet enkel interessant voor het wiskundige begrip van kleinheid en grootheid. Bijvoorbeeld, deze cumulativiteitsrelatie beschouwt alle instanties van het type *list A* in verschillende universa als zijnde (oordeelgewijs) gelijk aan elkaar. Deze nieuwe cumulativiteitsrelatie is nu geïntegreerd in Coq vanaf de officiële uitgave van Coq 8.7.

In het tweede deel van dit proefschrift ontwikkelen we modellen gebouwd met de techniek genaamd *logische relaties*, als operationeel gebaseerde semantische werkwijze voor typesysteemcorrectheid en voor het bewijzen van gelijkwaardigheid van programma's, voor een aantal programmeertalen. We ontwikkelen logischerelatiemodellen voor $\mathsf{F}_{\mu,ref,conc}$, een meerdradige (*concurrent*) programmeertaal die lijkt op ML. We maken gebruik van deze logischerelatiemodellen, om typesysteemcorrectheid te bewijzen voor $\mathsf{F}_{\mu,ref,conc}$, en ook om gelijkwaardigheid van programma's te bewijzen. We gebruiken het laatste model om te bewijzen dat twee meerdradige implementaties van datastructuren gelijkwaardig zijn. Een van de belangrijkste resultaten van dit proefschrift is de ontwikkeling van een logischerelatiemodel waarmee we de behoorlijke inkapseling van toestand (geheugen) kunnen tonen voor de programmeertaal STLang. STLang is een programmeertaal met een ST-monade, zoals in Haskell, die voor inkapseling van de toestand gebruikt kan worden. We tonen behoorlijke inkapseling van toestand door te bewijzen dat specifieke programma's gelijkwaardig zijn die niet gelijkwaardig zouden zijn als de toestand niet goed ingekapseld was. Dit probleem was een onopgelost probleem gedurende ongeveer twee decennia. Ten laatste ontwikkelen we logischerelatiemodellen voor een uitbreiding van $\mathsf{F}_{\mu,ref,conc}$ met continuaties, een mechanisme van sommige programmeertalen waarmee we de uitvoering van programma's kunnen opschorten en op een later moment hervatten. Het is welbekend dat continuaties redeneren over programma's ingewikkeld maken. We ontwikkelen een redeneerprincipe waarmee we het onverstorend gebruik van continaties kunnen negeren. Dus kunnen we redeneren over een programma dat geen verstorend gebruik maakt van continuaties net alsof er geen continuaties zijn in de programmeertaal. We gebruiken ons nieuwe redeneerprincipe samen met ons logischerelatiemodel om te bewijzen dat een implementatie van een *web server* waarin continuaties worden gebruikt (zoals in *web servers* geschreven in Racket) gelijkwaardig is aan een implementatie die geen gebruik maakt van continuaties.

Het is welbekend dat het ontwikkelen en gebruik maken van logischerelatie-modellen voor geavanceerde typesystemen zoals die die we bestuderen in de tweede deel van dit proefschrift, ingewikkeld is. We lossen dit probleem op in dit proefschrift door met Iris te werken. Iris is een grensverleggende nieuwe programmalogica gebaseerd op scheidingslogica (*separation logic*) die gericht is op het verifëren van meerdradige imperatieve programma's van hogere orde.

Door met Iris te werken kunnen we een directe behandeling van ingewikkelde aspecten van ons model vermijden. Daarenboven hebben we gebruik gemaakt van de implementatie van Iris in Coq om alle ontwikkelde theorieën van dit deel van het proefschrift in Coq te formaliseren.

Contents

List of Figures

Chapter 1

Introduction

Computer systems have been quite prolific in the past couple of decades; a trend that is undoubtedly going to accelerate even further. We rely on software systems and online services for virtually everything: banking, stock marketing, control systems of vehicles and planes, self-driving cars, TV sets, microwave ovens, etc. Failure in some of these technologies, e.g., the control system of a passenger plane, would be catastrophic and incur hefty human and financial costs. Hence, it is of utmost importance to ensure that safety- and security-critical software systems are correct.

One of the most important concepts that help us produce and verify correct software is the concept of a type system. Types play a crucial role in both computer science and mathematics. In computer science types enable compilers to check programs for (some) programmer's mistakes. In mathematics they form the basis of a group of proof assistants including Coq, Agda, etc. Proof assistants are programs that assist a mathematician by checking correctness of mathematical proofs including proofs of correctness of programs.

In this thesis we contribute to the study of programming languages and type theory. In the first part we formalize category theory, a mathematical theory closely related to the theory of types and programming languages, in the proof assistant Coq. Furthermore, we extend the underlying logic (type theory) of Coq to facilitate formalization and application of mathematical theories like category theory. In the second part, we study a number of properties of programs and programming languages through their types. This includes showing correctness of type systems (type soundness), showing how a more efficient implementation

of concurrent data structures refines less efficient versions[1] and establishing correct encapsulation of state by a Haskell-style ST monad. The ST monad allows a limited use of memory (state) while keeping programs pure. That is, programs behave as though they are not using memory, i.e., the use of state is encapsulated.

In the sequel we will give a brief explanation of types and how they are used for both formalization of mathematics (in proof assistants) and software development. This is followed by an overview of the contributions presented in the thesis.

1.1 Types

Types and type systems are very important notions in both computer science and mathematics. Statically typed programming languages use types to ensure that user-written programs are meaningful. An example of a meaningless program is a program that attempts to subtract a string of characters from a number, e.g., $10 - \text{“ABC”}$. Common sense dictates that such a statement is meaningless essentially because 10 is a number and “ABC” is a string of characters and the operation of subtraction is expected to be applied to two numbers. In the jargon of programming languages and type systems we say that $10 : \mathbb{Z}$ (read as: 10 has type $\mathbb{Z}$) and $\text{“ABC”} : \texttt{String}$ while $(-) : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$. The last statement simply states that the subtraction operation $(-)$ takes two integers and produces an integer. This information allows the compiler processing the program to reach the conclusion that the program $10 - \text{“ABC”}$ is meaningless and the result of a programmer's error.

Broadly and intuitively speaking, types describe what their members are. A program of type $\mathbb{Z}$ is an integer numeral, or rather it is a program that computes an integer numeral. Elements of the type $\mathbb{Z} \to \mathbb{Z}$ are functions that given an integer produce (after some computation) another integer. In general, types can be far more expressive. In the most extreme case, i.e., in a proof assistant like Coq, users can define their own types and one can define a type such that members of the type are mathematical proofs of a certain mathematical theorem.

Types for programming Every programming language features a number of basic types like $\mathbb{Z}$ (integers), $\mathbb{B}$ (Boolean), 1 (the unit type with a single

[1] Program e refines program e' if e can be used instead of e' as part of a bigger program without changing the overall behavior.

inhabitant () : 1), `String` (strings), etc. Apart from these virtually all programming languages feature what we call simple types (as in simply-typed λ-calculus). That is, for any two types τ and τ' we have the type of their Cartesian product, $\tau \times \tau'$, the type of their sum (also known as a tagged union), $\tau + \tau'$, and the function type, $\tau \to \tau'$.

Most advanced programming languages also feature types like recursive types, $\mu X.\, \tau$, references $\texttt{ref}(\tau)$ and polymorphic types $\forall X.\, \tau$. Recursive types are used to define recursive concepts, e.g., algebraic lists (encoded as $\mathit{list}_\tau \triangleq \mu X.\, 1 + (\tau \times X)$) where a list is either empty (uniquely defined as the unit type) or consists of a head (an element of τ) and a tail which is also again a list. The elements of the reference type $\texttt{ref}(\tau)$ are memory locations that always (guaranteed by the type system) store values of type τ. Polymorphic types, à la System F, are used to give types to polymorphic (generic) programs that can work with any types. For instance, the type of the identity function in System F is written as $\forall X.\, X \to X$. The programming languages that we study in the second part of this thesis feature all these types. Each chapter defines precisely the programming language that it studies and gives (excerpts of the important parts of) the syntax, operational semantics and typing rules of this programming language.

Types for mathematics The idea of using a type system as a logic goes back to Curry (1934) and Howard (1980) and is widely known as the Curry-Howard correspondence. This idea can be summarized as "propositions as types, proofs as programs". Curry (1934) and Howard (1980) noticed that there is a correspondence between types of the simply-typed λ-calculus and logical formulas of the intuitionistic propositional logic and between programs of a type and proofs of the proposition represented by that type. The following table shows the correspondence between the logical connectives of the intuitionistic propositional logic and types of the simply-typed λ-calculus.

Logical connective	Type
$\wedge$	$\times$
$\vee$	$+$
$\Rightarrow$	$\to$
$\top$	1
$\bot$	0

Intuitively, a proof of $P \wedge Q$ is a pair of proofs of P and Q. A proof of $P \vee Q$ is either a proof of P or a proof of Q. A proof of $P \Rightarrow Q$ is function that given a proof of P returns a proof of Q. The proposition $\top$ has a unique proof and

thus corresponds to the unit type. The absurd proposition $\bot$ has no proof and corresponds to the absurd type 0 which also has no inhabitant.

The idea of the Curry-Howard correspondence is generalized by Howard (1980) and de Bruijn (1970). This generalized version which is sometimes referred to as the Curry-Howard-de-Bruijn correspondence extends the idea of "propositions as types, proofs as programs" to predicate logic by extending simple types to dependent types. In dependent type theory types are also terms and themselves have types (usually called sorts or universes). This allows types to mention terms as in a type `Le` n m which depends on two natural numbers n and m and is only inhabited (is provable) if $n \leq m$. The calculus of constructions (CoC) (Coquand and Huet, 1988) and its extensions which are the basis of the proof assistant Coq are such dependent type theories.

In order to avoid paradoxes like Girard's paradox[2], arising from a type being term of itself, dependent type theories usually have a countable hierarchy of sorts.[3] In Coq this hierarchy is $\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 : \cdots$. Coq's type theory is *cumulative* dependent type theory which means, if we have that $A : \texttt{Type}_i$, then $A : \texttt{Type}_j$ whenever $i \leq j$. In addition to the hierarchy of universes $\texttt{Type}_i$ Coq also features a sort `Prop` of propositions. This is to separate programs (whose types are in some sort $\texttt{Type}_i$) from logical propositions (including properties of those program).[4]

Inductive types are important tools in formalizing mathematics in type theory. Inductive types allow us to define constructions that are inductive in nature which are ubiquitous in mathematics, e.g., (Peano's axiomatization of) natural numbers. For instance the type `Le` above corresponding to the $\leq$ relation on natural numbers can be defined as follows using the syntax of the Coq proof assistant.

```
Inductive Le (n : nat) : nat → Prop :=
| Le_n : Le n n
| Le_S : ∀ m, Le n m → Le n (S m).
```

This definition defines for each `n : nat` a family of predicates `Le n : nat → Prop` which is constructed as follows: (1) the proposition `Le n n` has a proof (`Le_n n`) and (2) if the proposition `Le n m` is inhabited then so is `Le n (S m)` by `Le_S n m`.

[2] Similar to Russel's paradox in set theory.

[3] The alternative is to have a finite hierarchy where the top sort is not itself a term of any type.

[4] Coq also features a universe `Set` which is just a shorthand for $\texttt{Type}_0$ as far as concerns this thesis.

1.2 Type theory and formalization of mathematics

Chapter 3 presents the formalization of a mathematical theory called category theory in the proof assistant Coq. This development encompasses most concepts and features of basic category theory, i.e., not including higher category theory or enriched categories. As such it should be useful as a basic mathematical library that can be used as a basis for other category-theoretical developments, e.g., formalizing categorical semantics of type theories. The other contribution of this development is the novel representation of smallness and largeness through type-theoretic universes which we will discuss below briefly.

A category is a collection of objects together with a collection of morphisms (also known as arrows) from A to B for any pair of objects A and B that need to satisfy certain axioms. The concept of a category is very general and hence many mathematical structures form categories. As a result, category theory can be used to give unifying and general definitions for common mathematical constructions. For instance the general concept of Cartesian product of two objects, the sum (tagged union) of two objects, the concept of a function space, etc. The archetypal example of a category is the category of sets where objects are sets and morphisms are functions between them. This category, usually written as **Set**, plays an important and central role in category theory. Categories are very closely related to type systems and logics (Jacobs, 1999; Lambek and Scott, 1988).

Categories are so general that they form a category themselves. The category of *small* categories has as objects small categories.[5] Note the qualifier *small* in the last sentence. Crucially, the category of small categories is itself *large* and is hence not an object of itself as it would be contradictory otherwise. In the formalization of category theory that is presented in Chapter 3 we use Coq's universes to formalize the concept of (relative) largeness and smallness. That is, we use a feature of Coq called universe polymorphism (introduced in Coq 8.5) to formally define categories. Universe polymorphism allows us to define a concept at all universes at once. That way, the category of categories is also defined at all universe levels and includes only categories as objects that are themselves at a *strictly* smaller universe level.[6] In Chapter 3 we argue that using universe levels to represent smallness and largeness works quite well in practice and only suffers from one caveat: categories are not cumulative. That is, a small category is *not* also a large category as is expected. This, as we shall argue in more

[5] The morphisms of this category are called functors but they are not relevant to the present discussion.

[6] This way of using type-theoretic universes to represent smallness and largeness is similar to the usual way of defining smallness and largeness in set theory using the closely related concept of Grothendieck universes. This is explained in more details in Chapter 3.

details in Chapter 3, has undesirable theoretical and practical consequences. The problem boils down to the fact that the cumulativity (subtyping) relation that is defined for sorts and dependent function types in the predicative calculus of inductive constructions (pCIC), the underlying type theory of Coq, is not extended to inductive types through which we define categories.

Chapter 3 is published in the proceedings of the first conference on formal structures for computation and deduction (FSCD'16) (Timany and Jacobs, 2016a). FSCD is a continuation (as of 2016) of the international conference on rewriting techniques and applications (RTA) which is ranked A by the Conference ranking portal of CORE.

Chapter 4 introduces the predicative calculus of cumulative inductive constructions (pCuIC) which extends the cumulativity relation in pCIC to inductive types. This new system forms the underlying type system of the proof assistant Coq as of the official release of version 8.7. The type system pCuIC adds a rule to pCIC that determines when an inductive type is a subtype of another inductive type. In the Coq proof assistant this allows us to determine for each universe-polymorphic inductive type I when I instantiated with some universe arguments is a subtype of another instance of I instantiated with some other universe arguments. For the case of categories the subtyping relation determined in pCuIC corresponds precisely to smallness and largeness of categories. The usefulness of the cumulativity for inductive types introduced in pCuIC is not limited to mathematical constructions like categories that involve smallness and largeness. Chapter 4 presents examples of subtyping of other inductive types like the universe-polymorphic definition of lists. We furthermore discuss how the new cumulativity relation extends Coq's template polymorphism feature which allows two instances of a *non-universe-polymorphic* inductive type in two different universes to be unified under certain conditions. Finally, in Chapter 4 we prove consistency of pCuIC as a logic by constructing a set-theoretic model of pCuIC in the Zermelo-Fraenkel set theory with the axiom of choice (ZFC) together with an extra axiom regarding existence of certain strongly inaccessible cardinals which are used to model type-theoretic universes in set theory. This model is based on and inspired by the model of Lee and Werner (2011) for pCIC.

Preliminary results of the constructions presented in Chapter 4 were published in the proceedings of the 12^{th} international colloquium on theoretical aspects of computing (ICTAC'15) (Timany and Jacobs, 2015). ICTAC is ranked B by the Conference ranking portal of CORE.

This chapter is accepted for publication in the third conference on formal structures for computation and deduction (FSCD'18) (Timany and Sozeau, 2018). FSCD is a continuation (as of 2016) of the international conference on

rewriting techniques and applications (RTA) which is ranked A by the Conference ranking portal of CORE.

1.3 Logical relations: a semantic approach to the study of programming languages through their types

The most important theorem that one proves about a type system of a programming language is *type soundness*, also known as *type safety*. This theorem states that types do fulfil their ultimate purpose, i.e., well-typed programs have well-defined behavior. In other words, the type soundness theorem states that well-typed programs are safe, i.e., never crash, which is usually summarized into the slogan "well-typed programs cannot go wrong" (Milner, 1978). This problem has been studied using denotational semantics (Milner, 1978) and syntactic approaches (Harper, 1994; Wright and Felleisen, 1994). The advantage of approaches based on denotational semantics over syntactic approaches is that the former are *modular* (separate parts can be proven safe separately) and take into account *data abstraction*. Syntactic approaches on the other hand can only be applied to whole programs that are well-typed and hence cannot be used to reason about programs where a part is not syntactically well-typed but can be proven to be safe otherwise. Furthermore, syntactic approaches only guarantee that the program does not crash and not that it will respect abstraction barriers between different parts of the program. For a further detailed discussion of the comparison of these two techniques see the Introduction section of Chapter 5.

In this thesis we take a third approach to this problem known as operationally based logical relations (Pitts, 1996). In this semantics approach we define for each type τ in the type system (by recursion on the structure of types) a predicate $[\![\tau]\!]$ such that for any program e, $[\![\tau]\!](e)$ implies that e is a safe program, $\mathit{Safe}(e)$, that *behaves* (based on the operational semantics) like a program of type τ. Note that behaving as a program of a type that involves abstract data types requires the program to respect these abstractions. In addition, since the logical relations model is defined semantically it is also modular. For instance, if e is a program that behaves as a function of type $\tau \to \tau'$ and e' is a program that behaves as a program of type τ then the program $e\ e'$ must behave as a program of type τ'. We prove the type soundness theorem using logical relations models by showing that all well-typed programs do fall in the predicate corresponding to their types and hence are safe.

Another important problem in program verification in particular and in the study of programming languages in general is proving equivalence of programs. Two programs are considered equivalent if replacing one by the other in any context, i.e., as part of any program, would not change the overall behavior of the program. This notion, called *contextual equivalence*, is the standard notion of equivalence of programs. This notion is important for showing the correctness of refactorings and optimizations. For instance, when in a big software project we replace an implementation of the queue abstract data structure with another more efficient implementation the overall behavior of the program should not change. This can be proven by showing that the two implementations are contextually equivalent.

Operationally based logical relations can also be used to develop methods to prove contextual equivalence of programs. In this case, instead of defining a predicate for each type we define a binary relation for each type. That is, $[\![\tau]\!]$ is a binary relation and for any pair of programs e and e' being in the relation $[\![\tau]\!](e, e')$ implies e and e' are programs of type τ and that e contextually refines e'. Contextual refinement is the one sided version of contextual equivalence, i.e., e refines e' if replacing e' by e cannot be detected by any context. Such a logical relations model can be used to prove contextual equivalence of programs by showing that each program refines the other.

It is well-known (Ahmed, 2004) that constructing and using logical relations models for programming languages with advanced types, e.g., dynamically allocated higher-order references, recursive types, polymorphism, etc., is difficult. Constructing these logical relations models requires using techniques known as step-indexing and recursive Kripke worlds. Furthermore, using these models is complicated by the fact that we need to take into account intricate details of the model, e.g., step-indices (the number of steps of execution of the program). In solving this problem we follow the idea of Plotkin and Abadi (1993) and Dreyer, Ahmed, and Birkedal (2009) and define our logical relations models in a logic that comes equipped with reasoning principles to reason about intricate details of the model at a high level of abstraction. The logic that we use for this purpose is the Iris program logic, a state-of-the-art program logic for verification of higher-order concurrent imperative programs. The logic of Iris is expressive enough that we can directly define our logical relations models in it without having to worry about intricate details of the model. Furthermore, working in Iris allows us to take advantage of its formalization and dedicated proof mode (Krebbers, Timany, and Birkedal, 2017) in the proof assistant Coq to formalize our logical relations models and their application in Coq. In particular, all the results presented in Chapters 5, 6 and 7 are formalized in the proof assistant Coq using Iris as a library.

In Chapter 5 we construct unary and binary logical relations models for

$\mathsf{F}_{\mu,ref,conc}$, a concurrent ML-like programming language featuring recursive types, dynamically allocated higher-order references, polymorphism and concurrency. We use the unary logical relations model to prove type safety of this language and use the binary logical relations model to prove equivalence of programs. In particular, we prove contextual refinement for a pair of concurrent counter implementations and a pair of concurrent stack implementations. In both of these cases we show that the more efficient version using fine-grained concurrency refines the version that uses a lock to protect the data structure. Fine-grained concurrent data structures do not use locks. They perform computations locally and only update the data structure atomically with the result of the computation. Hence, they do not block other threads from accessing the data structure while they are performing computations locally.

The results presented in Chapter 5 have been announced as part of a publication in the proceedings of the 44^{th} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17) (Krebbers, Timany, and Birkedal, 2017). POPL is ranked A^* by the Conference ranking portal of CORE.

Chapter 6 studies monadic encapsulation of state in STLang, a programming language featuring a Haskell-style ST monad. The ingenious idea of the ST monad introduced by Launchbury and Peyton Jones (1994) is simple yet elegant. It divides the memory (state) into phantom regions and uses types to annotate which region of memory an effectful command is using. The monadic type $\mathtt{ST}\ \tau\ \rho$ is the type of an effectful computation that when run produces a result of type τ and only uses the region of memory associated with ρ. The command $\mathtt{runST}\ \{e\}$ evaluates the effectful program e if it can be run in any region, i.e., if e has type $\forall X.\, \mathtt{ST}\ \tau\ X$. Since the effectful program being run can work in any region it can in particular work in an empty region of memory. Therefore, intuitively, the execution of effectful programs cannot be affected by the state when they start working. They can only use the part of memory that they allocate and initialize themselves. Hence, the memory is properly encapsulated and thus programs of STLang behave as though they are pure, i.e., as though they do not use memory at all. The problem that we address in Chapter 6 is how to formally prove that the ST monad does indeed properly encapsulate state. In that chapter we construct a logical relations model that allows to prove equivalence of certain programs that would not hold if the state was not properly encapsulated. Proving the equivalences that we prove in Chapter 6 was an open problem for about two decades.

Chapter 6 and its appendix are published in the proceedings of the 45^{th} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18) (Timany, Stefanesco, Krogh-Jespersen, and Birkedal, 2018). POPL is ranked A^* by the Conference ranking portal of CORE.

In Chapter 7 we work with a programming language $\mathsf{F}^{\mu,ref}_{conc,cc}$ which is an extension of $\mathsf{F}_{\mu,ref,conc}$ with continuations. Continuations allow programs to be suspended. Suspended programs (called continuations) can be resumed at a later point in time. In this chapter we develop logical relations models for $\mathsf{F}^{\mu,ref}_{conc,cc}$ to prove type soundness and to establish equivalence of programs. Continuations make reasoning about programs more intricate as we cannot consider programs in isolation anymore and need to consider the context under which programs are running as the context could be captured in a continuation. In this chapter we introduce so-called context-local reasoning principles which allow us to ignore innocuous usages of continuations. That is, we can reason about parts of programs that do not use continuations in a disruptive manner as though there are no continuations in the programming language. We use these context-local reasoning principles together with our binary logical relations model to prove that a web server that uses continuations to store client's state is equivalent to a traditional implementation that stores the state directly.

Remark Chapters of this thesis correspond to publications, already published or under submission. The bodies of these chapters are thus copied verbatim from these papers and are simply adjusted to adhere to the format of the present thesis. In particular, in the second part each chapter studies a different programming language which is precisely defined in that chapter. Since all chapters in the second part use Iris each of them gives an explanation of *the parts of* the Iris program logic that are used in that chapter.

Chapter 2

Preliminaries

In this chapter we give a cursory overview of the preliminaries necessary for reading this thesis.

2.1 Formal systems

The formal systems that we use in this thesis are the Coq proof assistant and the program logic Iris. The program logic Iris is used in the second part of the thesis. Each chapter in that part gives a short introduction to the parts of Iris that are used in that chapter. In this section we will give a brief general overview of the proof assistant Coq. For further details bout the Coq proof assistant and the Iris program logic, more developed examples and tutorials please refer to the following list of suggested reading:

For Coq

- Tutorials:
    1. Pierce, Amorim, Casinghino, Gaboardi, Greenberg, Hriţcu, Sjöberg, and Yorgey (2017)
    2. Huet, Kahn, and Paulin-Mohring (2018)
    3. Nahas (2012)
    4. Jacobs (2013)

- Coq reference manual: The Coq development team (2018)
- Books:
  1. Bertot and Castéran (2013)
  2. Chlipala (2013)
- Official website: `https://coq.inria.fr`

For Iris

- Scientific papers published in Conferences and journals:
  1. Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer (2015)
  2. Jung, Krebbers, Birkedal, and Dreyer (2016)
  3. Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017)
  4. Jung, Krebbers, Jourdan, Bizjak, Birkedal, and Dreyer (2018)
- Iris lecture notes: Birkedal and Bizjak (2017)
- Official website: `http://iris-project.org`

2.1.1 Coq

The system Coq in its most essential form consists of sorts (universes), dependent function types and inductive types.[1] All other constructs in Coq are (or can at least be thought of) as syntactic sugar defined in terms of these basic building blocks. The proof assistant Coq also includes many features that facilitate development of programs and proofs that we do not discuss here, e.g., type inference, type classes, canonical structures, the program system (the Russell language), etc.

Universes Universes (sorts) in Coq consist of `Prop` and the universe hierarchy $\texttt{Type}_0, \texttt{Type}_1, \cdots$. The sort `Set` is just a name for $\texttt{Type}_0$.[2]

[1] In this thesis we ignore coinductive types in Coq.

[2] Notice that Coq can be run with the option `-impredicative-set` which makes the sort `Set` impredicative. With this option the sort `Set` is no longer merely $\texttt{Type}_0$. However, this is beyond the scope of this thesis.

Dependent functions In our formalism of Coq in Chapter 4 we write $\lambda x : A.\, M$ for a (dependent) function. Dependent functions are sometimes referred to as dependent products. This term has type $\mathbf{\Pi} X : A.\, B$ if the body of the function, M, has type B given that x has type A. In Coq's notation we write `fun x : A ⇒ M` for the aforementioned dependent function and write `forall x : A, B` for the corresponding type. We call these functions dependent because the variable `x` can appear freely in `B` making the codomain type of the function depend on the value that the function is applied to. In particular, if `f` has type `forall x : A, B` and `t` has type `A` then the application `f t` will have type `B`$[$`t`$/$`x`$]$.

As a simple yet instructive example of a dependent type and a dependent function consider the definition of the identity function `id` and its type `Id`.

```
Definition Id := forall x : Type, x → x.
Definition id : Id := fun T z ⇒ z.
```

Note that the `A → B` is syntactic sugar for *non-dependent* function type. That is, `A → B` is a syntactic sugar for `forall x : A, B` for some `x` that does not appear freely in `B`. In the Coq code above we have written `Type` as the type of `x` and not explicitly a specific universe in the hierarchy, e.g., `Type@{i}`. Coq can automatically infer the universe levels of definitions and these are treated differently depending on which constructions are universe-polymorphic or universe-monomorphic. For a detailed discussion about treatment of universes see Chapter 4.

Inductive types, pattern matching and recursive functions The quintessential example of inductive types is that of Peano natural numbers. In Coq this type can be defined as follows:[3]

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

Here, `O` is zero and `S` is the successor function, intuitively, mapping n to $n + 1$.

The basic principle for working with inductive types is destructing them using the `match` construct. For instance, we can define the function `is_zero` to check whether a natural number is zero or not:

```
Definition is_zero (n : nat) : Bool :=
match n with
| O ⇒ true
| S _ ⇒ false
end.
```

[3] This is how the standard library of Coq defines Peano natural numbers.

In Coq one can define functions by recursion on a term of an inductive type.[4]

```
Fixpoint add (n m : nat) : nat :=
match n with
| O ⇒ m
| S k ⇒ S (add k m)
end.
```

Intuitively, if the first argument of `add` is zero then the result is the second argument. On the other hand, if the first argument is the successor of `k` then the result is the successor of the result of `add k m`. Notice that when a recursive call happens (the successor case) the first argument *strictly* decreases. This is crucial for ensuring consistency of Coq.

Coq supports dependent pattern matching. That is, the type of the result of the pattern matching on terms of inductive types can depend on the term being destructed. For details about the syntax and semantics of dependent pattern matching please consult the further reading materials given above.

Theorems and proofs Coq is a proof assistant and as such it has facilities to support interactive proofs. In Coq a theorem (respectively a lemma) starts with the keyword `Theorem` (respectively `Lemma`). This keyword is followed by the name of the theorem and the type of the theorem which represents the theorem that is to be proved (refer to Curry-Howard correspondence in Chapter 1).

The following piece of Coq code first loads the library of natural numbers from the standard library of Coq. Afterwards, it states a theorem expressing commutativity of the addition operation on natural numbers. The proof script below given between keywords `Proof.` and `Qed.` (see tactics and their use for proving theorems in the reading material for Coq given above) is a very simple and elaborated proof of this theorem type checked with Coq 8.7. The so-called bullets − and + are simply put to structure the proof by focusing on different cases in the proof (different cases of inductions here) and are optional parts of the proof script.

```
Require Import Coq.Init.Nat.

Theorem add_comm : forall n m : nat, add n m = add m n.
Proof.
  induction n; simpl in *.
  - induction m; simpl in *.
    + trivial.
    + rewrite ← IHm; trivial.
```

[4] This is how addition is defined in the standard library of Coq.

```
  - induction m; simpl.
    + rewrite IHn; trivial.
    + rewrite ← IHm, IHn. simpl.
      rewrite IHn; trivial.
Qed.
```

2.2 Programming languages studied in the second part

In the second part of this thesis we study programs and programming languages through their types. The programming languages studied in that part feature certain advanced features, e.g., dynamically allocated higher-order references, recursive types, etc. Most of these features are standard and are present in modern programming languages, e.g., OCaml. Here, we briefly explain these features insofar as to give examples of and explain the intuition for these features. Each chapter will introduce (the relevant parts of) the precise notation, syntax and semantics of the programming language that is studied in that chapter. Below, we will use the more-straightforward-to-read syntax of OCaml programming language for examples unless we need to explicitly mention the specific syntax of the programming languages that we study.

Higher-order code The programming languages that we study in the second part of this thesis are all higher-order programming languages. In a higher-order programming language, functions are first class values and can be passed as values to other functions. Functions that take as arguments other functions are called higher-order functions. The best example of such function is the `map_to_string` function that given a list of integers and a function from integers to strings produces a list of strings by applying the given function to all the elements of the given list.

```
let rec map_to_string l f =
match l with
| [] -> []
| h :: t -> (f h) :: (map_to_string t f)
```

Higher-order state Here, by state we mean the heap of the programming language. A programming language featuring higher-order references is a programming language where the state (heap) is higher-order. That is, alongside (first order) data, e.g., strings, integers, etc., one can also store (higher-order) functions in the heap. The idea here is to have references that store functions.

Different parts of the program can then read these references and use the stored function while other parts can change the stored function, effectively changing the behavior of the program dynamically.

The quintessential example of higher-order reference is how they can be used to implement recursive functions without any explicit use of recursion. In the following program the function `recursive_fun` takes a function as an argument and turns that function into a recursive function. The function in question must itself take a function (which is it will when it wants to make a recursive call). This program creates a reference for the recursive call which is initially the identity function. Subsequently, we update the reference storing the recursive call to the function that calls `f` with the stored recursive call. This example in the literature is known as the Landin's knot due to Landin (1964).

```
let recursive_fun (f : (int -> int) -> (int -> int)) =
  let recursive_call =
    ref (fun (x : int) -> x)
  in
  let rec_fun x =
    f !recursive_call x
  in
  recursive_call := rec_fun; rec_fun
```

The higher-order function `recursive_fun` can be used to encode the factorial function as follows:

```
let fact =
  let fact_rec rc n =
    if n = 0 then 1 else n * rc (n-1)
  in
  recursive_fun fact_rec
```

Impredicative polymorphism All programming languages studied in the second part of this thesis feature impredicative polymorphism, also known as parametric polymorphism. This feature allows us to write programs that are polymorphic in their types. That is, we can give types to programs that work with any value of any type. The quintessential example of polymorphic function is the polymorphic definition of the identity function.

```
let id : 'a -> 'a = fun x => x
```

Notice that the `'` in `'a` is to indicate that the type `'a` is the type variable of the polymorphic definition.

We call this form of polymorphism *impredicative* because the type variable of the polymorphic definition can be instantiated with any type, including the polymorphic type itself. That is, the expression `id id` has the type `'a -> 'a`.

There is a well-known problem with the combination of references and impredicative polymorphism. We cannot consider arbitrary expressions of the program polymorphic. If we were to do so then the expression `ref (fun x -> x)` could be typed `('a -> 'a) ref`. That is, according to this type, this expression results in a reference that for any type `t` has `(t -> t) ref`. For an argument of why allowing such typings violates type safety see Wright (1995) where the aforementioned example is taken from. The solution that Wright (1995) offers is called *value restriction* where only values, *and not arbitrary expressions*, can be considered polymorphic. The ML family programming languages use (variants of) the value restriction.

In order for us to ensure type-safety of the programming languages that we consider in the second part of this thesis we use a so-called type-level lambda which we write as Λ. The net effect on the operational semantics of the programming language is that the expression $\Lambda\, e$ is a value and cannot be evaluated further unless it is specialized at a specific type using the syntax $(\Lambda\, e)\ \_$. Our typing rules only allow expressions of the form $\Lambda\, e$ to be considered polymorphic. Such expressions by our definition are values. Hence, by construction our programming languages satisfy the value restriction conditions.

Recursive types Recursive types are used to represent recursive algebraic data structures such as lists. We write recursive types as $\mu X.\, \tau$. The type of lists with elements of type τ, $list_\tau$, is written as follows:

$$list_\tau \triangleq \mu X.\, 1 + (\tau \times X)$$

The idea here is that recursive types are defined as fixpoints such that $\mu X.\, \tau \simeq \tau[\mu X.\, \tau/X]$ where $\simeq$ is isomorphism. We have two language constructs for realization of this isomorphism in our programming languages, `fold` and `unfold`. This can be evidently seen in the typing rules associated with these two constructs:

$$\text{T-FOLD}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau/X]}{\Xi \mid \Gamma \vdash \texttt{fold}\, e : \mu X.\, \tau} \qquad \text{T-UNFOLD}\ \frac{\Xi \mid \Gamma \vdash e : \mu X.\, \tau}{\Xi \mid \Gamma \vdash \texttt{unfold}\, e : \tau[\mu X.\, \tau/X]}$$

See Chapter 5 for an explanation of these typing rules. In order to enable computation with values of recursive types the semantics of the programming language specifies that unfolding a folded value is that value itself, i.e., `unfold` (`fold` v) reduces to v.

The value *nil*, the empty list, the *cons* function, and the *head* function, taking the head of a list, can be written as follows in our programming languages:

$nil \triangleq$ `fold` (`inj`$_1$ ())

$cons \triangleq \lambda x.\ \lambda y.\ \texttt{fold}\,(\texttt{inj}_2\,(x, y))$

$head \triangleq \lambda x.\ \texttt{match}\,(\texttt{unfold}\,x)\,\texttt{with}\,\texttt{inj}_1\,y \Rightarrow \texttt{inj}_1\,() \mid \texttt{inj}_2\,y \Rightarrow \texttt{inj}_2\,(\pi_1\,y)\,\texttt{end}$

These programs have the following types for any type τ:

$$nil : list_\tau$$

$$cons : \tau \to list_\tau \to list_\tau$$

$$head : list_\tau \to 1 + \tau$$

Fine-grained concurrency We say a concurrent algorithm is coarse-grained if it uses a lock to protect the whole data structure. A fine-grained concurrent algorithm on the other hand does not use any locks. These algorithms, read the state of the data structure that they manipulate and perform the manipulation on their local copy of the data stored in that data structure. Afterwards, these algorithms, check that the data structure still stores the data that they had originally read. If so, they will update the data structure with their local manipulated version of the data. Crucially, this checking and updating happens *atomically* through the compare and set (`CAS`) instruction. Figure 5.5 in Chapter 5 shows both fine-grained and coarse-grained implementations of a concurrent counter module side-by-side.

Part I

Type Theory and Formalization of Mathematics

Chapter 3

Category Theory in Coq

This chapter is published in the proceedings of the first conference on formal structures for computation and deduction (FSCD'16) (Timany and Jacobs, 2016a).

We report on our experience implementing category theory in Coq 8.5. Our work formalizes most of basic category theory, including concepts not covered by existing formalizations, in a library that is fit to be used as a general-purpose category-theoretical foundation.

Our development particularly takes advantage of two features new to Coq 8.5: primitive projections for records and universe polymorphism. Primitive projections allow for well-behaved dualities while universe polymorphism provides a relative notion of largeness and smallness. The latter is one of the main contributions of this paper. It pushes the limits of the new universe polymorphism and constraint inference algorithm of Coq 8.5.

In this paper we present in detail smallness and largeness in categories and the foundation they are built on top of. We furthermore explain how we have used the universe polymorphism of Coq 8.5 to represent smallness and largeness arguments by simply ignoring them and entrusting them to the universe inference algorithm of Coq 8.5. We also briefly discuss our experience throughout this implementation, discuss concepts formalized in this development and give a comparison with a few other developments of similar extent.

3.1 Introduction

A category (Awodey, 2010; Mac Lane, 1978) consists of a collection of objects and for each pair of objects A and B a collection of morphisms (aka arrows or homomorphisms) from A to B. Moreover, for each object A we have a distinguished morphism $id_A : A \to A$. Morphisms are composable, i.e., given two morphisms $f : A \to B$ and $g : B \to C$, we can compose them to form: $g \circ f : A \to C$. Composition must satisfy the following additional conditions: $\forall f : A \to B.\ f \circ id_A = f = id_B \circ f$ and $\forall f, g, h.\ (h \circ g) \circ f = h \circ (g \circ f)$.

The notion of a category can be seen as a generalization of sets. In fact sets as objects together with functions as morphisms form the important category **Set**. On the other hand, it can be seen as a generalization of the mathematical concept of a preorder. In this regard, a category can be thought of as a preorder where objects form the elements of the preorder and morphisms from A to B can be thought of as "witnesses" of the fact that $A \preceq B$. Thus, identity morphisms are witnesses of reflexivity whereas composition of morphisms forms witnesses for transitivity and the additional axioms simply spell out coherence conditions for witnesses. Put concisely, categories are preorders where the quality and nature of the relation holding between two elements is important. In this light, categories are to preorders what intuitionistic logic is to classical

logic. A combination of these two interpretations of categories can provide an essential and useful intuition for understanding most, if not all, of the theory.

This generality and abstractness is what led some mathematicians to call this mathematical theory "general abstract nonsense" in its early days. However category theory, starting from this simple yet vastly abstract and general definition, encompasses most mathematical concepts and has found applications not only in mathematics but also in other disciplines, e.g, computer science.

In computer science it has been used extensively, especially in the study of semantics of programming languages (Mitchell, 1996), in particular constructing the first (non-trivial) model of the untyped lambda calculus by Dana Scott (Salibra, 2012), type systems (Jacobs, 1999), and program verification (Biering, Birkedal, and Torp-Smith, 2007; Birkedal, Mogelberg, Schwinghammer, and Stovring, 2011; Birkedal, Støvring, and Thamsborg, 2010).

Given the applications of category theory and its fundamentality on the one hand and the arising trend of formalizing mathematics in proof assistants on the other, it is natural to have category theory formalized in one; in particular, a formalization that is practically useful as a category-theoretical foundation for other works. This paper is a report of our experience developing such a library. There already exist a relatively large number of formalizations of category theory in proof assistants (Ahrens, Kapulkin, and Shulman, 2015; Gross, Chlipala, and Spivak, 2014a; Huet and Saïbi, 2000; Megacz, 2011; Peebles, Deikun, Norell, Doel, Vezzosi, Jahandarie, and Cook, 2016). However, most of these implementations are not general purpose and rather focus on parts of the theory which are relevant to the specific application of the authors. See the bibliography of Gross, Chlipala, and Spivak (2014b) for an extensive list of such developments.

Features of Coq 8.5 used: η for records and universe polymorphism This development makes use of two features new to Coq 8.5. Namely, primitive projection for records (i.e., the η rule for records) and universe polymorphism.

Following Gross, Chlipala, and Spivak (2014a), we use primitive projections for records which allow for well behaved-dualities in category theory. The dual (aka opposite) of a category $\mathcal{C}$ is a category $\mathcal{C}^{op}$ which has the same objects as $\mathcal{C}$ where the collection of morphisms from A to B is swapped with that from B to A. Drawing intuition from the similarity of categories and preorders, the opposite of a category (seen as a preorder) is simply a category where the order of objects is reversed. Use of duality arguments in proofs and definitions in category theory are plentiful, e.g., sums and products, limits and co-limits, etc. One particular property of duality is that it is involutive. That is, for any

category $\mathcal{C}$, $(\mathcal{C}^{op})^{op} = \mathcal{C}$. The primitive projection for records simply states that two instances of a record type are definitionally equal if and only if their projections are. In terms of categories, two categories are definitionally equal if and only if their object collections are, morphism collections are and so forth. This means that we get that the equality $(\mathcal{C}^{op})^{op} = \mathcal{C}$ is definitional. Similar results hold for the duality and composition of functors, for natural transformations, etc. That is we get definitional equalities such as $(\mathcal{F}^{op})^{op} = \mathcal{F}$, $(\mathcal{N}^{op})^{op} = \mathcal{N}$ and $(\mathcal{F} \circ \mathcal{G})^{op} = \mathcal{F}^{op} \circ \mathcal{G}^{op}$ where $\mathcal{F}$ and $\mathcal{G}$ are functors and $\mathcal{N}$ is a natural transformation.

To achieve well behaved dualities, in addition to primitive projections one needs to slightly adjust the definition of a category itself. More precisely, the definition of the category must carry a symmetric form of associativity of composition. The reason being the fact that for the dual category we can simply swap the proof of associativity with its symmetric form and thus after taking the opposite twice get back the proof we started with.

In this development we have used universe polymorphism, a feature new to Coq 8.5, to represent relative smallness/largeness. In short, universe polymorphism allows for a definition to be polymorphic in its universe variables. This allows us, for instance, to construct the category of (relatively small) categories directly. That is, the category constructed is at a universe level (again polymorphic) while its objects are categories at a lower universe level. We will elaborate the use of universe polymorphism to represent relative largeness and smallness below in Section 3.2.

Contributions

The main contributions of this development are its extent of coverage of basic concepts in category theory and its use of the universe polymorphism of Coq 8.5 and its universe inference algorithm to represent relative smallness/largeness. The latter, as explained below, allows us to represent smallness and largeness using universe levels by simply forgetting about them and letting Coq's universe inference algorithm take care of smallness and largeness requirements as necessary.

The structure of the rest of this paper

The rest of this paper is organized as follows. Section 3.2 gives an explanation of smallness and largeness in category theory based on the foundation used. This is followed by a detailed explanation of our use of the new universe polymorphism

and universe constraint inference algorithm of Coq 8.5 to represent relative smallness/largeness of categories. There, we also give a short comparison of the way other developments represent (relatively) large concepts.

In Section 3.3, we give a high-level explanation of the concepts formalized and some notable features in this work. We furthermore provide a comparison of our work with a number of other works of similar extent. We also briefly discuss the axioms that we have used throughout this development.

Section 3.4 describes the work that we have done or plan to do which is based on the current work as category-theoretical foundation. Finally, in Section 3.5 we conclude with a short summary of the paper.

Development source code The repository of our development can be found in GitHub (Timany, 2016a).

3.2 Universes, Smallness and Largeness

A category is usually called small if its objects and morphisms form sets and large otherwise. It is called locally small if the morphisms between any two objects form a set but objects fail to. For instance, the category **Set** of sets and functions is a locally small category as the collection of all sets does not form a set while for any two sets, there is a set of functions between them. These distinctions are important when working with categories. For instance, a category is said to be complete if it has the limit of all *small* diagrams ($\mathcal{F} : \mathcal{C} \to \mathcal{D}$ is a small diagram if $\mathcal{C}$ is a small category). For instance, **Set** is complete but does not have the cartesian product of all large families of sets.

These terminology and considerations are due to the fact that the original foundations of category theory by Eilenberg and Mac Lane were laid on top of NBG (von Neumann-Bernays-Gödel) set theory. In NBG, in addition to sets, the notion of a class (a collection of sets which itself is not *necessarily* a set) is also formalized. For any property φ, there is a class C_φ of all sets that have property φ. If the collection of sets satisfying φ forms a set then C_φ is just that set. Otherwise, C_φ is said to be a proper class. In this formalism, one can formalize large categories but cannot use them. For instance, the functor category $\mathbf{Set}^{\mathbf{Set}}$ is not defined as its objects are already proper classes and there is no class of proper classes in NBG.

The other foundation that is probably the most popular among mathematicians is that of ZF with Grothendieck's axiom of universe. Roughly speaking, a Grothendieck universe V is a set that satisfies ZF axioms, e.g., if $A \in V$ and

$B \in V$ then $\{A, B\} \in V$ (axiom of pairing), if $A \in V$ then $2^A \in V$ (axiom of power set), etc. We also have if $A \in B$ and $B \in V$ then $A \in V$. *Grothendieck's axiom* says that for any set x there is a Grothendieck universe V such that $x \in V$. This also implies that for any Grothendieck universe V, there is a Grothendieck universe V' such that $V \in V'$.

Working on top of this foundation, one can talk about V-small categories and use all the set-theoretic power of ZF. The notion of completeness for a V-small category can be defined as having all V-small limits. The category of all V-small sets will be a V'-small category where $V \in V'$. It is also a V-locally-small category as its set of morphisms are V-small but its set of objects fail to be. For more details on foundations for category theory see chapter 12 of McLarty (1996).

The type hierarchy of Coq (also known as universes), as explained below, bears a striking resemblance to Grothendieck universes just explained. In the rest of this section we discuss how Coq's new universe polymorphism feature allows us to use Coq universes instead of Grothendieck universes in a completely transparent way. That is, we never mention any universes in the whole of the development and Coq's universe inference algorithm (part of the universe polymorphism feature) infers them for us.

3.2.1 Coq's Universes

In higher-order dependent type theories such as that of Coq, types are also terms and themselves have types. As expected, allowing existence of a type of all types results in self-referential paradoxes, such as Girard's paradox (Coquand, 1986). Thus, to avoid such paradoxes type theories like Coq use a countably infinite hierarchy of types of types (also known as universes): $\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 : \ldots$ The type system of Coq additionally has the cumulativity property, i.e., for any term $T : \texttt{Type}_0$ we also have $T : \texttt{Type}_{n+1}$.

The type system of Coq has the property of *typical ambiguity*. That is, in writing definitions, we don't have to specify universe levels and/or constraints on them. The system automatically infers the constraints necessary for the definitions to be valid. In case, the definition is such that no (consistent) set of constraints can be inferred, the system rejects it issuing a "universe inconsistency" error. It is due to this feature that throughout this development we have not had the need to specify any universe levels and/or constraints by hand.

To better understand typical ambiguity in Coq, let's consider the following definition.

```
Definition Tp := Type.
```

In this case, Coq introduces a new universe variable for the level of the type `Type`. That is, internally, the definition looks like[1]:

```
Definition Tp : Type@{i+1} := Type@{i}.
```

Note that in older version of Coq and when universe polymorphism is not enabled in Coq 8.5 the universe level `i` above is a global universe level, i.e., it is fixed. Hence, the following definition is rejected with a universe inconsistency error.

```
Definition TpTp : Tp := Tp.
```

The problem here is that this definition requires(`Type@{i} : Type@{i}`) which requires the system to add the constraint `i < i` which makes the set of constraints inconsistent. Without universe polymorphism, one way to solve this problem would be to duplicate the definition of `Tp` as `Tp'` which would be internally represented as:

```
Definition Tp' : Type@{j+1} := Type@{j}.
```

Now we can define `TpTp'`:

```
Definition TpTp' : Tp' := Tp.
```

which Coq accepts and consequently adds the constraint `i < j` to the global set of universe constraints. As these constraints are global however, after defining `TpTp'` we can't define `Tp'Tp`

```
Definition Tp'Tp : Tp := Tp'.
```

This is rejected with a universe inconsistency error as it requires `j < i` to be added to the global set of constraints which makes it inconsistent as it already contains `i < j` from `TpTp'`.

3.2.2 Universe Polymorphism

Coq has recently been extended (Sozeau and Tabareau, 2014) to support universe polymorphism. This feature is now included in the recently released Coq 8.5. When enabled, universe levels of a definition are bound at the level of that definition. Also, any universe constraints needed for the definition to be well-defined are local to that definition. That is the definition of `Tp` defined above is represented internally as:

```
Definition Tp@{i} : Type@{i+1} := Type@{i}. (* Constraints: *)
```

[1] `Type@{i}` is Coq's syntax for $\mathtt{Type}_i$.

Note that the universe level `i` here is local to the definition. Hence, `Tp` can be instantiated at different universe levels. As a result, the definition of `TpTp` above is no longer rejected and is represented internally as:

```
Definition TpTp@{i j} : Tp@{j} := Tp@{i}. (* Constraints: i < j *)
```

That is, the two times `Tp` is mentioned, two different instances of it are considered at two different universe levels `i` and `j` resulting in the constraint $\mathtt{i} < \mathtt{j}$ for the definition to be well-defined.

Note the resemblance between universes in Coq and Grothendieck universes, e.g., cumulativity or axiom of pairing, i.e., if `A : Type@{i}` and `B : Type@{i}` then $\{\mathtt{x} : \mathtt{Type@\{i\}} \mid \mathtt{x} = \mathtt{A} \vee \mathtt{x} = \mathtt{B}\} : \mathtt{Type@\{i\}}$, etc.

In the sequel, in some cases, we only show the internal representation of concepts formalized in Coq.

3.2.3 Smallness and Largeness

In this implementation, we use universe levels as the underlying notion of smallness/largeness. In other words, we simply ignore smallness and largeness of constructions and simply allow Coq to infer the necessary conditions for definitions to be well-defined. We define categories without mentioning universe levels. They are internally represented as:

```
Record Category@{i j} :=
 {
   Obj : Type@{i};
   Hom : Obj → Obj → Type@{j};
   ...
 } : Type@{max(i+1, j+1)} (* Constraints: *)
```

The category of (small) categories is internally represented as:

```
Definition Cat@{i j k l} :=
 {|
   Obj := Category@{k l};
   Hom := fun (𝒞 𝒟 : Category@{k l}) ⇒ Functor@{k l k l} 𝒞 𝒟;
   ...
 |} : Category@{i j}
(* Constraints: k < i, l < i, k ≤ j, l ≤ j *)
```

That is, **Cat** has as objects categories that are small compared to itself.

Having a universe-polymorphic **Cat** means for any category $\mathcal{C}$ there is a version of **Cat** that has $\mathcal{C}$ as an object. Therefore, for example, to express the fact

that two categories are isomorphic, we simply use the general definition of isomorphism in the specific category **Cat**. This means we can use all facts and lemmas proven for isomorphisms, for isomorphisms of categories with no further effort required.

The category of types (representation of **Set** in Coq) is internally represented as:

```
Definition Set@{i j} :=
  {|
    Obj := Type@{j};
    Hom := fun (A B : Type@{j}) ⇒ A → B;
    ...
  |} : Category@{i j} (* Constraints: j < i *)
```

The constraint `j < i` above is exactly what we expect as **Set** is locally small. The reason that Coq's universe inference algorithm produces this constraint is that the type of objects of **Set** is `Type@{j}` which itself has type `Type@{i}`. But, the homomorphisms of this category are functions between two types whose type is `Type@{j}`. Thus, the type of homomorphisms themselves is `Type@{j}`. For details of typing rules for function types see the manual of Coq (The Coq development team, 2015).

Complete Small Categories are Mere Preorder Relations! Perhaps the best showcase of using the new universe polymorphism of Coq to represent smallness/largeness can be seen in the theorem below which simply implies that any complete category is a preorder category, i.e., there is at most one morphism between any two objects.

```
Theorem Complete_Preorder (𝒞 : Category) (CC : Complete 𝒞) :
      forall x y : Obj 𝒞, Hom x y' ≃ ((Arrow 𝒞) → Hom x y)
```

where `y'` is the limit of the constant functor from the discrete category **Discr**(`Arrow` $\mathcal{C}$) that maps every object to `y`, `(Arrow` $\mathcal{C}$`)` is the type of all homomorphisms of category $\mathcal{C}$ and $\simeq$ denotes isomorphism. In other words, for any pair of objects `x` and `y` the set of functions from the set of all morphisms in $\mathcal{C}$ to the set of morphisms from `x` to `y` is isomorphic to the set of morphisms from `x` to some constant object `y'`. This though, would result in a contradiction as soon as we have two objects A and B in $\mathcal{C}$ for which the collection of morphisms from A to B has more than one element. Hence, we have effectively shown that *any* complete category is a preorder category.

This is indeed absurd as the category **Set** is complete and there are types in Coq that have more than one function between them! However, this theorem

holds for small (in the conventional sense) categories. That is, any *small and complete* category is a preorder category[2].

As expected, the constraints on the universe levels of this theorem that are inferred by Coq do indeed confirm this fact. That is, this theorem is in fact only applicable to a category $\mathcal{C}$ for which the level of the type of objects is less than or equal to the level of the type of arrows. This is in direct conflict with the constraints inferred for **Set** as explained above. Hence, Coq will refuse to apply this theorem to the category **Set** with a universe inconsistency error.

3.2.4 Limitations Imposed by Using Universe Levels for Smallness and Largeness

The universe polymorphism of Coq, as explained in Sozeau and Tabareau (2014), treats inductive types by considering copies of them at different levels. Furthermore, if a term of a universe polymorphic inductive type is assumed to be of two instances of that inductive type with two different sets of universe level variables, additional constraints are imposed so that the corresponding universe level variables in those two sets are required to be equal. As records are a special kind of inductive types, the same holds for them. For us, this implies that if we have $\mathcal{C}$: `Category@{i j}` and we additionally have that $\mathcal{C}$: `Category@{i' j'}`, Coq enforces `i = i'` and `j = j'`. This means, **Cat**`@{i j k l}` is in fact *not* the category of *all* smaller categories. Rather it is the category of smaller categories that are at level `k` and `l` and not any lower level.

Apart from the fact that **Cat** defined this way is not the category of all relatively small categories, these constraints on universe levels impose practical restrictions as well. For instance, looking at the fact that **Cat**`@{i j k l}` has exponentials (functor categories), we can see the constraints that `j = k = l`. Consequently, only those copies have exponentials for which this constraints holds. Looking back at **Set**, we had the constraint that the level of the type of morphisms is strictly less than that of objects. This means, there is no version of **Cat** that both has exponentials and a version of **Set** in its objects.

Moreover, we can use the Yoneda lemma to show that in any cartesian closed category, for any objects a, b and c:

$$(a^b)^c \simeq a^{b \times c} \tag{3.1}$$

Yet, this theorem can't be applied to **Cat**, even though it holds for **Cat**.

It is worth noting that although the category **Cat**`@{i j k l}` is the category of all categories `Category@{k l}` and not lower, for any lower category it contains

[2]This theorem and its proof are taken from Awodey (2010).

an "isomorphic copy" of that category. That is any category $\mathcal{C}$: `Category@{k' l'}` such that $\mathtt{k}' \leq \mathtt{k}$ and $\mathtt{l}' \leq \mathtt{l}$ can be "lifted" to `Category@{k l}`. Such a lifting function can be simply written as:

```
Definition Lift (C : Category@{k' l'}) : Category@{k l} :=
  {| Obj := Obj C; Hom := Hom C; ... |}.
```

and the appropriate constraints, i.e., $\mathtt{k}' \leq \mathtt{k}$ and $\mathtt{l}' \leq \mathtt{l}$ are inferred by Coq. However, working with such liftings is in practice cumbersome as in interesting cases where $\mathtt{k}' < \mathtt{k}$ and/or $\mathtt{l}' < \mathtt{l}$, we can't prove or even specify `Lift` $\mathcal{C} = \mathcal{C}$ as it is ill-typed. This means, any statement regarding $\mathcal{C}$ must be proven separately for `Lift` $\mathcal{C}$ in order for them to be useful for the lifted version.

It is possible to alleviate these problems if we have support for cumulative inductive types in Coq, as proposed in Timany and Jacobs (2015). In such a system, any category $\mathcal{C}$: `Category@{i j}` will also have type `Category@{k l}` so long as the constraints $\mathtt{i} \leq \mathtt{k}$ and $\mathtt{j} \leq \mathtt{l}$ are satisfied.

However, these limitations are not much more than a small inconvenience and in practice we can work in their presence with very little extra effort. At least as far as basic category theory goes. Our development is an attestation to that.

3.2.5 Smallness and Largeness in Other Developments

In homotopy type theory (HoTT) (The Univalent Foundations Program, 2013) a category $\mathcal{C}$ has a further constraint that for any two objects A and B the set of morphisms from A to B must form an hSet (a homotopy type-theoretical concept). On the other hand, for two categories $\mathcal{C}$ and $\mathcal{D}$, the set of functors from $\mathcal{C}$ to $\mathcal{D}$ does not necessarily form an hSet. It does however when the set of objects of $\mathcal{D}$ forms an hSet. Therefore, in HoTT settings one can construct the category of small *strict* categories, i.e., small categories whose type of objects forms an hSet, and not the category of all small categories. However, the category of small strict categories itself is not strict. Hence, contrary to the category **Cat** in our development, there is no category (in the HoTT sense, i.e., one whose objects form an hSet) that has the category of small strict categories as one of its objects. In this regard, working in HoTT is similar to working in NBG rather than ZF with Grothendieck universes.

The situation regarding the category of small strict categories discussed above is due to the fact that homotopy type-theoretical levels for types (e.g., hSet) concern a notion of (homotopy theoretical) *complexity* rather than cardinality. In fact, in other situations, e.g., in defining limits of functors, where cardinality is concerned universe levels can be used to express smallness and largeness. In other words, in HoTT settings, when defining limits, one can simply not

mention universe levels and let Coq infer that the definition of limit for a functor $\mathcal{F} : \mathcal{C} \to \mathcal{D}$ is well-defined whenever, $\mathcal{C}$ is relatively small compared to $\mathcal{D}$. This also means that the restrictions mentioned above are also present in HoTT settings when universe levels are used to represent smallness and largeness. For instance isomorphism 3.1 above can't be proven in **Cat** using the Yoneda lemma even if a, b and c are strict categories.

This is how smallness and largeness works in both Gross, Chlipala, and Spivak (2014a) and Ahrens, Kapulkin, and Shulman (2015). This is also the case for our development when ported on top of the HoTT library (*HoTT Version of Coq and Library*). As one consequence, contrary to what was explained above, in migrating to the HoTT library settings we can't simply consider the isomorphism of categories as the general notion of isomorphism in the specific case of **Cat**.

In Huet and Saïbi (2000), working in Coq 8.4, the authors define a duplicate definition of categories, `Category'`, tailored to represent large categories. This way, they form the `Category'` of categories (`Category`) – much like we used `Tp'` above.

Peebles, Deikun, Norell, Doel, Vezzosi, Jahandarie, and Cook (2016) however use universe levels to represent smallness and largeness. But working in Agda which provides no typical ambiguity or cumulativity, they have to hand code all universe levels everywhere; whereas we rely on Coq's inference of constraints to do the hard work. Noteworthy is also the fact that their categories have three universe variables instead of our two. One for the level of the type of objects, one for the level of the type of morphisms and one for the level of the type of the setoid equality for their setoids of morphisms.

3.3 Concepts Formalized, Features and Comparison

In this development we have formalize most of the basic category theory. Here, by basic we mean not involving higher (e.g., 2-categories), monoidal or enriched categories. This spans over the simple yet important and useful basic concepts like terminal/initial objects, products/sums, equalizers/coequalizers, pullbacks/pushouts and exponentials on the one hand and adjunctions, Kan extensions, (co)limits (as (left)right local Kan extensions) and toposes on the other.

The well-behaved dualities (in the sense discussed above) allow us to simply define dual notions, just as duals of their counterparts, e.g., initial objects as

terminal objects of the dual category or the local left Kan extension of $\mathcal{F}$ along $\mathcal{G}$ as the local right Kan extension of $\mathcal{F}^{op}$ along $\mathcal{G}^{op}$.

3.3.1 Concepts Formalized: Generality and Diversity

Throughout this development we have tried to formalize concepts in as general a way as possible so long as they are comfortably usable. For instance, we define (co)limits as (left)right local Kan extensions along the unique functor to the terminal category. By doing so, we can extend facts about them to (co)limits. As an example, consider (left)right adjoints preserving (co)limits and (co)limit functors being adjoint to Δ explained below.

Different versions of adjunction and Kan extensions In this formalization, we have multiple versions of the definition of adjunctions and Kan extensions. In particular, we define unit-universal morphism property adjunction, unit-co-unit adjunction, universal morphism adjunction and hom-functor adjunction. For these different versions, we provide conversions to and from the unit-universal morphism property definition which is taken to be the main definition. This definition is also taken to be the main definition of adjunction in Awodey (2010). For local Kan extensions, we define them as (initial)terminal (co)cones along a functor as well as through the hom-functor. Global Kan extensions are simply defined through adjunctions.

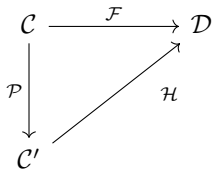
The main reason for this diversity, aside from providing a versatile category theory library, is the fact that each of these definitions is most suitable for some specific purpose.

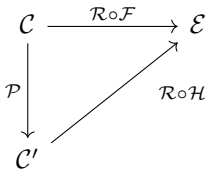
For instance, using the hom-functor definition of adjunctions makes it very easy to prove that isomorphic functors have the same adjoints: $\mathcal{F} \simeq \mathcal{F}' \Rightarrow \mathcal{F} \dashv \mathcal{G} \Rightarrow \mathcal{F}' \dashv \mathcal{G}$, duality of adjunction: $\mathcal{F} \dashv \mathcal{G} \Rightarrow \mathcal{G}^{op} \dashv \mathcal{F}^{op}$, and uniqueness of adjoint functors: $\mathcal{F} \dashv \mathcal{G} \Rightarrow \mathcal{F}' \dashv \mathcal{G} \Rightarrow \mathcal{F} \simeq \mathcal{F}'$. The last case simply follows from the Yoneda lemma. On the other hand, the unit-universal morphism property definition of adjunctions together with the definition of Kan extensions as cones along a functor provide an easy way to convert from local to global Kan extensions.

Universal morphism adjoints in practice express sufficient conditions for a functor to have a (left)right adjoint. That is, a functor $\mathcal{G} : \mathcal{C} \to \mathcal{D}$ is a right adjoint (has a left adjoint functor) if the comma category $(x \downarrow \mathcal{G})$ has a terminal object for any $x : \mathcal{D}$. As we will briefly discuss below, (left)right adjoint functors preserve (co)limits. Freyd's adjoint functor theorem gives an answer to the question "when is a functor that preserves all limits a right adjoint (has a left

adjoint functor)". Universal morphism adjoints appear in this theorem and that's why we have included them in our formalization.

(Left)right adjoints preserve (co)limits Awodey (2010) devotes a whole section to this fact with the title "RAPL" (Right Adjoints Preserve Limits). For a better understanding of this fact and perhaps the concept of adjunctions, let us draw intuition from categorical interpretations of logic. In categorical interpretations of logic, the existential and universal quantifiers are interpreted as left and right adjoints to some functor while conjunctions and disjunctions are defined as products and sums respectively which respectively are in turn limits and co-limits (see Jacobs (1999) for details). In this particular case, RAPL and its dual boil down to: $\forall x.\ P(x) \wedge Q(x) \Leftrightarrow \forall x.\ P(x) \wedge \forall x.\ Q(x)$ and $\exists x.\ P(x) \vee Q(x) \Leftrightarrow \exists x.\ P(x) \vee \exists x.\ Q(x)$. We prove this fact in general for (left)right local Kan extensions. To this end, the unit-co-unit definition of adjunctions is the easiest to use to prove the main lemma which along with hom-functor definition of Kan extensions proves that (left)right adjunctions preserve (left)right Kan extensions. That is for an adjunction $\mathcal{L} \dashv \mathcal{R}$ where $\mathcal{R} : \mathcal{D} \to \mathcal{E}$ and $\mathcal{L} : \mathcal{E} \to \mathcal{D}$ if in the diagram on the left $\mathcal{H}$ is the local right Kan extension of $\mathcal{F}$ along $\mathcal{P}$ then in the right diagram $\mathcal{R} \circ \mathcal{H}$ is the local right Kan extension of $\mathcal{R} \circ \mathcal{F}$ along $\mathcal{P}$:





The case of (co)limits follows immediately. In Coq we show this by constructing a local right Kan extension (using the hom-functor definition) of $\mathcal{R} \circ \mathcal{F}$ along $\mathcal{P}$ where the Kan extension functor (`HLRKE`) is $\mathcal{R}$ composed with the Kan extension functor of $\mathcal{F}$ along $\mathcal{P}$:

```
Definition Right_Adjoint_Preserves_Hom_Local_Right_KanExt
 {C C' : Category} (P : Functor C C') {D : Category} (F : Functor C D)
 (hlrke : Hom_Local_Right_KanExt P F) {E : Category}
 {L : Functor E D} {R : Functor D E} (adj : UCU_Adjunct L R)
: Hom_Local_Right_KanExt P (R ∘ F) :=
 {|
   HLRKE := (R ∘ (HLRKE hlrke));
   HLRKE_Iso := ...
 |}.
```

(Co)limit functors are adjoint to Δ In order to show that (co)limits are adjoint to the diagonal functor (Δ) we simply use the fact that local (left)right Kan extensions assemble together to form (left)right global Kan extensions. As global Kan extensions are defined as (left)right adjoints to the pre-composition functor, putting these two facts together, we effortlessly obtain that (co)limits form functors which are (left)right adjoint to Δ.

Cardinality restrictions We introduce the notion of cardinality restriction in the category **Set**. A cardinality restriction is a property over types (objects of **Set**) such that if it holds for some type, it must hold for any other type isomorphic (in **Set**) to it. That is, if a cardinality restriction holds for a type, it must hold for any other type with the same cardinality.

```
Record Card_Restriction : Type :=
 { Card_Rest : Type → Prop;
   Card_Rest_Respect : forall (A B : Type),
    (A ≃≃ B ::> Set) → Card_Rest A → Card_Rest B
 }.
```

The type (`A ≃≃ B ::> Set`) is the type of isomorphisms $A \simeq B$ in **Set**. As an example, the cardinality restriction corresponding to finiteness is defined as follows.

```
Definition Finite : Card_Restriction :=
 {| Card_Rest :=
    fun A ⇒ inhabited {n : nat & (A ≃≃ {x : nat | x < n} ::> Set)}; ...
 |}.
```

The definition above basically says that a type A is finite if there exists some n such that A is isomorphic to the type $\{x : \texttt{nat} \mid x < n\}$ of natural numbers less than n.

(Co)limits restricted by cardinality We use the notion of cardinality restrictions above to define (co)limits restricted by cardinality. For a cardinality restriction P, we say a category $\mathcal{C}$ has (co)limits of cardinality P ($\mathcal{C}$ is P-(co)complete) if for all functors $\mathcal{F} : \mathcal{D} \to \mathcal{C}$ such that $P(Obj_{\mathcal{D}})$ and $\forall AB \in Obj_{\mathcal{D}}, P(Hom(A, B))$, $\mathcal{C}$ has the (co)limit of $\mathcal{F}$.

```
Definition Has_Restr_Limits (C : Category) (P : Card_Restriction) :=
 forall {J : Category} (F : Functor J C),
  P J → P (Arrow J) → Limit F.
```

We state several lemmas about cardinality restricted (co)completeness, e.g., if a category has all limits of a specific cardinality its dual has all co-limits of that cardinality.

```
Definition Has_Restr_Limits_to_Has_Restr_CoLimits_Op
 {C : Category} {P : Card_Restriction}
 (HRL : Has_Restr_Limits C P) : Has_Restr_CoLimits (C^op) P := ...
```

This also allows us to define a topos, simply as a category that is cartesian closed, has all finite limits and a subobject classifier where finiteness is represented as a cardinality restriction.

```
Class Topos : Type :=
 { Topos_Cat : Category;
   Topos_Cat_CCC : CCC Topos_Cat;
   Topos_Cat_Fin_Limit : Has_Restr_Limits Topos_Cat Finite;
   Topos_Cat_SOC : SubObject_Classifier Topos_Cat
 }.
```

(Co)Limits by (Sums)Products and (Co)Equalizers A discrete category is a category where the only morphisms are identities. That is, any set can induce a discrete category by simply considering the category which has as objects members of that set and the only morphisms are identity morphisms. We define the discrete category of a type A as a category, $\mathbf{Discr}(A)$ with terms of type A as objects and the collection of morphisms from an object x to an object y are proofs of equality of $x = y$.

```
Definition Discr_Cat (A : Type) : Category :=
 {| Obj := A; Hom := fun a b ⇒ a = b; ... |}.
```

Similarly, a discrete functor is a functor that is induced from a mapping f from a type A to objects of a category $\mathcal{C}$:

```
Definition Discr_Func {C : Category} {A : Type} (f : A → C) :
  Functor (Discr_Cat A) C := {| FO := f; ... |}.
```

We define the notion of generalized (sums)products to be that of (co)limits of functors from a discrete category.

```
Definition GenProd {A : Type} {C : Category} (f : A → C) :=
 Limit (Discr_Func f).
```

We use these generalized (sums)products to show that any category that has all generalized (sums)products and (co)equalizers has all (co)limits. We also prove the special case of cardinality restricted (co)limits. Using the notions explained above, we show that given a cardinality restriction P if a category has (co)equalizers as well as all generalized (sums)products that satisfy P, then that category is P-(co)complete.

```
Definition Restr_GenProd_Eq_Restr_Limits
   {C : Category} (P : Card_Restriction)
```

```
    {CHRP : forall (A : Type) (f : A → C), (P A) → (GenProd f)}
    {HE : Has_Equalizers C}
: Has_Restr_Limits C P := ...
```

Categories of Presheaves To the best of our knowledge, ours is the only category theory development featuring facts about categories of presheaves such as their (co)completeness, and being a topos. The category of presheaves on $\mathcal{C}$, ($\mathbf{PSh}(\mathcal{C})$), is a category whose objects are functors of the form $\mathcal{C}^{op} \to \mathbf{Set}$ and whose morphisms are natural transformations. In other words, a presheaf $P : \mathcal{C}^{op} \to \mathbf{Set}$ on $\mathcal{C}$ is a collection of sets indexed by objects of $\mathcal{C}$ such that for a morphism $f : A \to B$ in $\mathcal{C}$, there is a function (a conversion if you will) $P(f) : P(B) \to P(A)$ in **Set**. Presheaves being toposes, each come with their own logic. As an example, Birkedal, Mogelberg, Schwinghammer, and Stovring (2011) show that the logic of the category of presheaves on ω (the preorder of natural numbers considered as a category) corresponds to the step-indexing technique used in the field of programming languages and program verification. For more details about elementary properties of categories of presheaves see Awodey (2010). There categories of presheaves are called categories of diagrams.

3.3.2 Comparison

Figures 3.1 and 3.2 give an overall comparison of our development with select other implementations of category theory of comparable extent. These figures mention only the most notable features and concepts formalized and do not contain many notions and lemmas in these developments. Notice also that the list of concepts and features appearing in these tables is by no means exhaustive and is not the union of all formalized concepts and features of these developments. In these figures, our development is the first column.

3.3.3 Axioms

One axiom that is used ubiquitously throughout the development is the uniqueness of proofs of equality.

```
forall (A : Type) (x y : A) (p q : x = y), p = q
```

We in practice enforce this axiom using proof-irrelevance (as `p` and `q` are proofs). To facilitate the use of this axiom, we prove a number of lemmas, e.g.:[3]

[3]This is an over-simplification: in practice types of `FA` $\mathcal{F}$ and `FA` $\mathcal{G}$ don't match and therefore their equality as stated here is ill-typed. In practice, we adjust the type of `FA` $\mathcal{F}$ using the equality of object maps.

Concept / Feature	[1]	[2]	[3]	[4]	[5]
Automation	partial	✓			
Based on HoTT	in [6]♯	✓		✓	
Setoid for Morphisms			✓		✓
Assumes UIP or equivalent	few restricted cases				✓
Basic constructions:					
Terminal/Initial object	✓	✓	✓	✓	✓
Products/Sums	✓		✓	✓	✓
Equalizers/Coequalizers	✓		✓		
Pullbacks/Pushouts	✓		✓	✓	✓
Basic constructions above are (co)limits	✓		✓		
exponentials	✓		✓		✓
Subobject classifier	✓			✓	✓
External constructions:					
Comma categories	✓	✓	✓	✓	✓
Product category	✓	✓	✓	✓	✓
Sum category		✓			
Cat. of categories (**Cat**):	✓	✓	✓		✓
Cartesian closure	✓		✓		
Initial/terminal object	✓	✓	✓		✓
Category of sets (**Set**):	✓	✓	✓	✓	✓
Basic (co)limits	✓	init./term.	partial		
(Local†)Cartesian closure	✓		CCC		
(Co†)Completeness	✓		comp.		✓
Sub-object classifier	(`Prop : Type`)†				
Topos	✓†				
Hom functor	✓	✓	✓	✓	✓
Fully-faithful functors	✓	✓		✓	✓
Essentially (inj)sur-jective functors	✓	✓		✓	✓
The Yoneda lemma	✓	✓	✓	✓	✓
Monoidal Categories		partial			✓
Enriched Categories		partial			partial
2-categories					✓
Pseudo-functors		✓			✓
(Co)monads and algebras :					
(Co)Monad				✓	✓
T-(co)algebras (T: an endofunctor)	✓			✓	✓
Eilenberg Moore cat.					✓
Kleisli cat.					✓

[1]: Timany (2016a); [2]: Gross, Chlipala, and Spivak (2014a); [3]: Huet and Saïbi (2000) [4]: Ahrens, Kapulkin, and Shulman (2015); [5]: Peebles, Deikun, Norell, Doel, Vezzosi, Jahandarie, and Cook (2016); [6]: Timany (2016b).
†Uses the axioms: propositional extensionality and constructive indefinite description (choice).
♯The version of our development we are migrating to HoTT settings, on top of HoTT library.

Figure 3.1: Comparison of features and concepts formalized with a few other implementations of comparable extent

Concept / Feature	[1]	[2]	[3]	[4]	[5]
Adjunction	✓	✓	✓		✓
Unit-universal morphism adjunction	✓	✓			
Hom-functor adjunction	✓	✓	✓		
Unit-counit adjunction	✓	✓	✓	✓	✓
Universal morphism adjunction	✓	✓	✓		
Uniqueness up to natural isomorphism	✓				
Naturally isomorphic functors have the same left/right adjoints	✓				
Adjoint composition laws	✓	✓			✓
Category of adjunctions (objects: categories; morphisms: adjunctions)	✓				
Partial adjunctions		✓			
Adjoint Functor Theorem	✓				✓
Kan extensions	✓	✓			✓
Global definition	✓	✓		✓	
Local definition	✓	✓			
Through hom-functor	✓				
Through cones (along a functor)	✓				✓
Through partial adjoints		✓			
Uniqueness	✓				
Preservation by adjoint functors	✓				
Naturally isomorphic functors form the same left/right Kan extension	✓				
Pointwise kan extensions (preserved by representable functors)	✓	✓			
(Co)Limits	✓	✓	✓	✓	✓
As (left)right kan extensions	✓	✓			
As (initial)terminal (co)cones			✓	✓	✓
(Sum)Product-(co)equalizer (co)limits	✓				
(Co)Limit functor	✓	✓			
(Co)Limits functor adjoint to Δ	✓	✓			
(Co)limits restricted by cardinality	✓				
Pointwise (as kan extensions), i.e., preserved by Hom functor	✓		✓		
Category of presheaves over $\mathcal{C}$ ($\mathbf{PSh}_{\mathcal{C}}$):	✓				✓
Terminal/Initial object	✓				
Products/Sums	✓				
Equalizers/Coequalizers	✓†				
Pullbacks	✓				
Cartesian closure	✓				
Completeness/Co-completeness	✓†				
Sub-object classifier (Sieves)	✓†				
Topos	✓†				

[1]: Timany (2016a); [2]: Gross, Chlipala, and Spivak (2014a); [3]: Huet and Saïbi (2000) [4]: Ahrens, Kapulkin, and Shulman (2015); [5]: Peebles, Deikun, Norell, Doel, Vezzosi, Jahandarie, and Cook (2016); [6]: Timany (2016b).
†Uses the axioms: propositional extensionality and constructive indefinite description (choice).

Figure 3.2: Comparison of features and concepts formalized with a few other implementations of comparable extent (cont.)

```
Lemma Functor_eq_simplify (𝒞 𝒟 : Category) (ℱ 𝒢 : Functor 𝒞 𝒟) :
      (FO ℱ = FO 𝒢) → (FA ℱ = FA 𝒢) → ℱ = 𝒢
```

which says two functors are equal if their object and arrow maps are. If so, the proofs that the arrow maps preserve identity and composition are just assumed equal using proof-irrelevance (uniqueness of equality proofs).

Using uniqueness of equality proofs in the definition of categories is an essential necessity. As otherwise, as explained in the HoTT book (The Univalent Foundations Program, 2013), the category defined is not a category but a form of higher category. That's why in any formalization of category theory this axiom is assumed or enforced in one way or another.

In homotopy type theory (HoTT) settings, assuming uniqueness of proofs of equality in general is in direct contradiction with the univalence axiom which sits at the heart of HoTT. Therefore in developments of category theory on top of HoTT, e.g., Gross, Chlipala, and Spivak (2014a) and Ahrens, Kapulkin, and Shulman (2015), they include the fact that proofs of equalities of morphisms are unique as part of the definition of a category. This is precisely the requirement that collections of morphisms should form hSets discussed above.

In developments using setoids, e.g., Huet and Saïbi (2000) and Peebles, Deikun, Norell, Doel, Vezzosi, Jahandarie, and Cook (2016), the authors customize the setoid equalities so that proofs are never considered. For instance, they *define* the setoid equality for functors so that two functors are equal whenever their object and morphism maps are.

We are currently in the process of porting a version of our development on top of the HoTT library[4]. There we also stop using this axiom and change the definition of categories. As expected almost all of the cases where we use uniqueness of proofs of equality (in a direct or indirect way) are not problematic in HoTT settings, i.e., they are applied to equality of morphisms. However, there are a few limited cases were they are not. Some of these cases are no longer relevant in the HoTT settings and some others are very easily surmountable. For more details of our ongoing effort of porting this development on top of the HoTT library see the extended version of this paper (Timany and Jacobs, 2016b).

Apart from the axiom of uniqueness of proofs of equality, we have made frequent use of the axiom of functional extensionality. However, this axiom is a consequence of the univalence axiom and is in fact provided in the HoTT library and frequently used therein.

[4]The version being ported on top of the HoTT library can be found at GitHub (Timany, 2016b).

We have in particular taken advantage of two other axioms, propositional extensionality and axiom of choice (constructive indefinite description in the library of Coq) which we have used, e.g., to construct co-limits in **Set** and presheaf categories. Along with using setoids, using these axioms to represent quotient types in type theory is standard practice. We plan to use higher inductive types, as explained in the HoTT book (The Univalent Foundations Program, 2013), to construct such co-limits in the version ported on top of the HoTT library.

3.4 Future Work: Building on Categories

We believe that this development is one that provides a foundation for other works based on category-theoretical foundations. We have plans to make use of the foundation of category theory that has been laid in this work. In particular, we plan to make use of this foundation for mechanization of categorical logic (see Jacobs (1999)) and higher order separation logic (see Biering, Birkedal, and Torp-Smith (2007)) for the purpose of using them as foundations for mechanization of program verification. In particular, the theory of presheaves developed provides a basis for formalization of the internal logic of presheaf categories with a particular interest in the topos of trees (Birkedal, Mogelberg, Schwinghammer, and Stovring, 2011).

In this regard, we have already used this development as a foundation to formalize the theory of Birkedal, Støvring, and Thamsborg (2010) to solve category theoretical recursive ultra-metric space equations (Timany and Jacobs, 2016c). In Birkedal, Støvring, and Thamsborg (2010), the authors use the theory of ultra-metric spaces to build unique (up to isomorphism) fixed-points of particular category-theoretical recursive domain-theoretic equations. More precisely, they construct fixed-points of a particular class of mixed variance functors, i.e., functors of the form $\mathcal{F} : (\mathcal{C}^{op} \times \mathcal{C}) \to \mathcal{C}$. Solutions to such mixed-variance functors can for example be used to construct models for imperative programming languages. Successful implementation of this theory (Timany and Jacobs, 2016c) on top of our general foundation of categories, although arguably not huge, is evidence that this development is fit for being used as a general-purpose foundation.

In Birkedal, Støvring, and Thamsborg (2010), the authors define the notion of an M-category to be a category in which the set of morphisms between any two objects form a non-empty ultra-metric space. In our formalization, based on a general theory of ultra-metric spaces, we define M-categories as categories in which the type of morphisms between any two objects forms an

ultra-metric space, dropping the rather strong non-emptiness requirement. We instead require some weaker conditions which still allow us to form fixed-points.

An interesting instance of M-categories is the presheaf topos of the preorder category of natural numbers, i.e., the topos of trees. In our development, just showing that this category qualifies as an M-category is sufficient to immediately be able to construct desired fixed-points. This is due to the fact that in the foundations provided, all necessary conditions for an M-category to allow formation of solutions, e.g., existence of limits of a particular class of functors is already established.

3.5 Conclusion

The most important conclusion of this paper is that Coq 8.5 with its new features: η for records and universe polymorphism, is next to ideal for formalization of category theory and related parts of mathematics. We believe that Coq 8.5 is the first version of Coq that makes it possible to lay a truly useful and versatile general purpose category theoretical foundation as we have demonstrated.

In summary, we surveyed our development of the foundations of category theory. This development features most of the category-theoretical concepts that are formalized in most other such developments and some more. We pushed the limits of the new feature of universe polymorphism and the constraint inference algorithm of Coq 8.5 by using them to represent relative smallness/largeness. As discussed, it gives very encouraging results despite the restrictions imposed by not having cumulative inductive types.

We have successfully used this implementation as the categorical foundation to build categorical ultra-metric space theoretic fixed-points of recursive domain equations. This seems an encouraging initial indication that this work is fit to perform the important role of a general purpose category theoretical foundation for other developments to build upon.

Acknowledgements We would like to thank Jason Gross, Matthieu Sozeau and Georges Gonthier for helpful discussions. We are also grateful to the anonymous reviewers for their invaluable comments and suggestions.

This work was funded by EU FP7 FET-Open project ADVENT under grant number 308830.

Table of symbols (pCuIC)

Γ	Typing context
Δ	List of declarations
$t[u/x]$	Substitution
$t[\vec{u}/\vec{x}]$	Simultaneous substitution
$\Gamma \vdash t : A$	Typing judgement
$\mathbf{\Pi} x : A.\ B$	Dependent function type
$A \to B$	Non-dependent function type
`Prop`	Universe `Prop`
`Set`	Universe `Set`
$\texttt{Type}_i$	i^{th} Universe
ℓ	level
$\mathcal{L}$	Set of levels
$\mathbb{U}_\ell$	Algebraic universe at level ℓ
$=$	Equality of mathematical objects, e.g, sets
$\equiv$	Syntactic equality of two terms
$\simeq$	Judgemental equality
$\preceq$	Subtyping/Cumulativity relation
$\mathcal{WF}$	Wellformedness of typing contexts
$\mathsf{Ind}_n \{\Delta_I := \Delta_C\}$	Mutually inductive block
$\mathcal{D}$	A mutually inductive block
Q	Motive of elimination

$\mathbf{Elim}(t; \cdots ; \cdots) \{\cdots\}$	Elimination of inductive types
$\xi_{\mathcal{D}}^{\vec{Q}}(\cdot, \cdots)$	Type of case eliminator
$\mathcal{V}_\alpha$	Von Neumann cumulative hierarchy at stage α
κ	Strongly inaccessible cardinal
$[\![\Gamma \vdash t]\!]$	Interpretation of $\Gamma \vdash t$ in the model
$\Pi a \in A.\, B(a)$	Set of set-theoretic dependent functions
Lam	Trace encoding of functions
App	Application of trace encoded function
Φ	A rule set (for set theoretic inductive definitions)
$\frac{A}{a}$	A rule (for set theoretic inductive definitions)
$\mathcal{I}(\Phi)$	Fixpoint of a rule set
$A{\downarrow}$	A is defined

Chapter 4

Cumulative inductive types in Coq

This chapter is accepted for publication in the proceedings of the third conference on formal structures for computation and deduction (FSCD'18) (Timany and Sozeau, 2018).

In order to avoid well-known paradoxes associated with self-referential definitions, higher-order dependent type theories stratify the theory using a countably infinite hierarchy of universes (also known as sorts), $\mathtt{Type}_0 : \mathtt{Type}_1 : \cdots$. Such type systems are called cumulative if for any type A we have that $A : \mathtt{Type}_i$ implies $A : \mathtt{Type}_{i+1}$. The predicative calculus of inductive constructions (pCIC) which forms the basis of the Coq proof assistant, is one such system.

In this paper we present and establish the soundness of the predicative calculus of cumulative inductive constructions (pCuIC) which extends the cumulativity relation to inductive types. We discuss cumulative inductive types as present in Coq 8.7 and their application to formalization and definitional translations.

4.1 Introduction

In higher-order dependent type theories every type is a term and hence has a type. As expected, having a type of all types which is a term of its own type, leads to inconsistencies such as Girard's paradox (Girard, 1972) and Hurken's paradox (Hurkens, 1995). To avoid this, a predicative hierarchy of universes is usually employed. The predicative Calculus of Inductive Constructions (pCIC) at the basis of the Coq proof assistant (The Coq Development Team, 2017), additionaly supports *cumulativity*: as a Pure Type System with subtyping, it includes the rule: $\Pi\Gamma.\mathtt{Type}_i \leq \Pi\Gamma.\mathtt{Type}_{i+1}$.

Earlier work (Sozeau and Tabareau, 2014) on universe-polymorphism in Coq allows constructions to be polymorphic in universe levels. The quintessential universe-polymorphic construction is the polymorphic definition of categories:

```
Record Category@{i,j} :=
  { Obj : Type@{i}; Hom : Obj → Obj → Type@{j}; ··· }.
```
[1]

However, pCIC does not extend the subtyping relation (induced by cumulativity) to inductive types. As a result, there is no subtyping relation between distinct instances of a universe-polymorphic inductive type. That is, for a category `C`, having both `C : Category@{i,j}` and `C : Category@{i',j'}` is only possible if $\mathtt{i} = \mathtt{i}'$ and $\mathtt{j} = \mathtt{j}'$.

In this work, we build upon the preliminary and in-progress work of Timany and Jacobs (2015) on extending pCIC to pCuIC (predicative Calculus of Cumulative Inductive Constructions). In pCuIC, subtyping of inductive types no longer imposes the strong requirement that both instances of the inductive type need to have the same universe levels. In addition, in pCuIC we consider two inductive

[1] Records in Coq are syntactic sugar for an inductive type with a single constructor. `Type@{i}` is Coq's syntax for $\mathtt{Type}_i$.

types that are in mutual cumulativity relation to be judgementally equal. This cumulativity relation is also extended to the constructors of inductive types, resulting in a very lax criteria for conversion of constructors. In pCuIC, in order for a term $\mathtt{C} : \mathtt{Category}_{\mathtt{i},\mathtt{j}}$ to have the type $\mathtt{Category}_{\mathtt{i}',\mathtt{j}'}$, *i.e.*, for the cumulativity relation $\mathtt{Category}_{\mathtt{i},\mathtt{j}} \preceq \mathtt{Category}_{\mathtt{i}',\mathtt{j}'}$ to hold, it is only required that $\mathtt{i} \leq \mathtt{i}'$ and $\mathtt{j} \leq \mathtt{j}'$. This is indeed what a mathematician would expect when universe levels of the type `Category` are thought of as representing (relative) smallness and largeness. For more details on representing relative size reasoning in category theory using universe levels see Timany and Jacobs (2016a).

Contributions Timany and Jacobs (2015) give an account of then work-in-progress on extending pCIC with a single cumulativity rule for cumulativity of inductive types. The authors show the soundness of a rather restricted subsystem of their system. In this paper, we extend and complete this work, through the following contributions:

- We extend Timany and Jacobs (2015) to support lowering levels as well as lifting them. For instance, given universe levels $i < j$ and a type $A : \mathtt{Type}_i$, the old system of Timany and Jacobs (2015) only allowed the subtyping $list_i\ A \preceq list_j\ A$. Our generalization of the subtyping relation for inductive types also allows $list_j\ A \preceq list_i\ A$ and furthermore judgementally equates them, *i.e.*, $list_i\ A \simeq list_j\ A$. Similarly for constuctors, it justifies $nil_i\ A \simeq nil_j\ A$, rendering universe annotations computationally irrelevant in this case.

- This generalization allows universe polymorphism to subsume the functionality of *template polymorphism*, a feature of Coq which allows under certain conditions two instances of a *non-universe-polymorphic* inductive type at different universe levels to be unified.

- We prove soundness of cumulativity by giving a model in ZFC which builds on the one of Lee and Werner (2011). This model naturally supports cumulativity for inductive types, as most set-theoretic models will. However, the argument for consistency in Lee and Werner (2011) assumes strong normalization to model recursive functions, which already implies consistency. We solve this problem by resorting to eliminators instead of the fixpoint and case constructs.

- Cumulativity of inductive types as presented in this paper is integrated in the stable version 8.7 of Coq (The Coq Development Team, 2017). We discuss remaining issues regarding the replacement of template polymorphism by universe polymorphism with cumulative inductive types.

$$\textsc{WF-ctx-hyp}\ \frac{\Gamma \vdash A : s \qquad x \notin \mathrm{dom}(\Gamma)}{\mathcal{WF}(\Gamma, x : A)} \qquad \textsc{WF-ctx-def}\ \frac{\Gamma \vdash t : A \qquad x \notin \mathrm{dom}(\Gamma)}{\mathcal{WF}(\Gamma, (x := t : A))} \qquad \textsc{Prop}\ \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \texttt{Prop} : \texttt{Type}_i}$$

$$\textsc{Hierarchy}\ \frac{\mathcal{WF}(\Gamma) \qquad i < j}{\Gamma \vdash \texttt{Type}_i : \texttt{Type}_j} \qquad \textsc{Let}\ \frac{\Gamma, (x := t : A) \vdash u : B}{\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u : B[t/x]}$$

$$\textsc{App}\ \frac{\Gamma \vdash M : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]} \qquad \textsc{Var}\ \frac{\mathcal{WF}(\Gamma) \qquad x : A \in \Gamma \text{ or } (x := t : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{App-eq}\ \frac{\Gamma \vdash M \simeq M' : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N \simeq N' : A}{\Gamma \vdash M\ N \simeq M'\ N' : B[N/x]}$$

$$\textsc{Prod}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \mathbf{\Pi} x : A.\, B : s_3}$$

$$\textsc{Lam}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \mathbf{\Pi} x : A.\, B : s}{\Gamma \vdash \lambda x : A.\, M : \mathbf{\Pi} x : A.\, B}$$

$$\textsc{Prod-eq}\ \frac{\Gamma \vdash A \simeq A' : s_1 \qquad \Gamma, x : A \vdash B \simeq B' : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \mathbf{\Pi} x : A.\, B \simeq \mathbf{\Pi} x : A'.\, B' : s_3}$$

Figure 4.1: An excerpt of the typing rules for the basic constructions

- We highlight two applications of Cumulative Inductive Types: one to the formalization of the Yoneda lemma, and the other one to the construction of definitional translations / syntactic models of type theories.

4.2 Predicative calculus of inductive constructions (pCIC)

In this section we give a short account of the system pCIC, presented with an equality judgment. Note that this system does not feature universe polymorphism. We will discuss universe polymorphism in Section 4.3. The full system pCIC can be found in Appendix A. The sorts of pCIC are as follows:

$\texttt{Prop}, \texttt{Set} = \texttt{Type}_0, \texttt{Type}_1, \texttt{Type}_2, \ldots$ We write the dependent product (function) type as $\mathbf{\Pi} x : A.\, B$. This is the type of functions that given $t : A$, produce a result of type $B[t/x]$. We write lambda abstraction in the Church style, $\lambda x : A.\, t$. The term $\textbf{let}\, x := t : A\, \textbf{in}\, u$ is the Church style let binding. We write function applications as juxtapositions, e.g., $M\ N$. Figure 4.1 shows an excerpt of the typing rules for these basic constructions.

There are three different judgements in this figure: well formedness of typing contexts $\mathcal{WF}(\Gamma)$, the typing judgement, $\Gamma \vdash t : A$, *i.e.*, term t has type A under the typing context Γ, and judgemental equality, $\Gamma \vdash t \simeq t' : A$, *i.e.*, terms t and t' are judgementally equal terms of type A under the typing context Γ. Most of the basic constructions (wherever it makes sense) come with a rule for judgemental equality. These rules indicate which parts of the constructions are sub-terms that can be replaced by some other judgementally equal term. For example, the rule Prod-eq states that the domain and codomain of (dependent) function types can be replaced by judgementally equal terms. The relation $\mathcal{R}_s(s_1, s_2, s_3)$ determines the sort of the product type based on the sort of the domain and codomain. The relation is defined as follows: $\mathcal{R}_s(\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\mathsf{max}\{i,j\}})$, $\mathcal{R}_s(\texttt{Prop}, \texttt{Type}_i, \texttt{Type}_i)$ and $\mathcal{R}_s(s, \texttt{Prop}, \texttt{Prop})$. Note that the impredicativity of the sort Prop is enforced by this relation.

Inductive types In this paper we consider blocks of mutual inductive types that live in predicative universes. Inductive types in Prop add extra complexity to the construction of set theoretic models. On the other hand, inductive types in Prop can be encoded using their Church encoding. For instance, the type False and conjunction of two predicates can be defined as follows:

```
Definition conj (P Q : Prop) := forall (R : Prop), (P → Q → R) → R.
Definition False := forall (P : Prop), P.
```

We write $\textbf{Ind}_n \{\Delta_I := \Delta_C\}$ for an inductive block where n is the number of parameters, Δ_I is list of inductive types of the block and Δ_C is the list of constructors. The arguments of an inductive type that are not parameters are known as *indices*. The following are some of the examples of inductive types written in this format: natural numbers, lists, vectors and a mutually inductive encoding of forests respectively.

$\textbf{Ind}_0\{nat : \texttt{Set} := Z : nat, S : nat \to nat\}$

$\textbf{Ind}_1\{list : \mathbf{\Pi} A : \texttt{Set}.\, \texttt{Set} := nil : \mathbf{\Pi} A : \texttt{Set}.\, list\ A,$

$\qquad cons : \mathbf{\Pi} A : \texttt{Set}.\, A \to list\ A \to list\ A\}$

$\textbf{Ind}_1\{vec : \mathbf{\Pi} A : \texttt{Set}.\, nat \to \texttt{Set} := vnil : \mathbf{\Pi} A : \texttt{Set}.\, vec\ A\ Z,$

$$\textsc{Ind-WF}\quad \frac{\begin{array}{c}\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) \\ (A \equiv \mathbf{\Pi} p : P.\, \mathbf{\Pi} m : M.\, A_d \qquad \Gamma \vdash A : s_d \text{ for all } (d : A) \in \Delta_I) \\ (T \equiv \mathbf{\Pi} p : P.\, T' \\ \Gamma, \Delta_I, p : P \vdash T' : A_d \text{ for all } (c : T) \in \Delta_C \text{ if } c \in \mathsf{Constrs}(\Delta_C, d))\end{array}}{\mathcal{WF}(\Gamma, \mathbf{Ind}_n\, \{\Delta_I := \Delta_C\})}$$

Assuming $\mathcal{D} \equiv \mathbf{Ind}_n\, \{\Delta_I := \Delta_C\} \in \Gamma$ and $\mathcal{WF}(\Gamma)$:

$$\textsc{Ind-type}\quad \frac{d_i \in \mathrm{dom}(\Delta_I)}{\Gamma \vdash \mathcal{D}.d_i : \Delta_I(d_i)} \qquad \textsc{Ind-constr}\quad \frac{c \in \mathrm{dom}(\Delta_C)}{\Gamma \vdash \mathcal{D}.c : \Delta_C(c)[\overrightarrow{\Delta_I.d}/\,\vec{d}]}$$

$$\textsc{Ind-Elim}\quad \frac{\begin{array}{c}\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n\, \{\Delta_I := \Delta_C\} \in \Gamma \\ \mathrm{dom}(\Delta_I) = \{d_1, \ldots, d_l\} \qquad \mathrm{dom}(\Delta_C) = \{c_1, \ldots, c_{l'}\} \\ (\Gamma \vdash Q_{d_i} : \mathbf{\Pi} \vec{x} : \vec{A}.\, (d_i\ \vec{x}) \to s' \text{ where} \\ \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{x} : \vec{A}.\, s \text{ for all } 1 \le i \le l) \\ \Gamma \vdash t : \mathcal{D}.d_k\ \vec{u}\ \vec{m} \qquad \mathsf{len}(\vec{u}) = n \\ \Gamma \vdash f_{c_i} : \xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i)) \text{ for all } 1 \le i \le l'\end{array}}{\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\, \{f_{c_1}, \ldots, f_{c_{l'}}\} : Q_{d_k}\ \vec{u}\ \vec{m}\ t}$$

Figure 4.2: Typing rules for inductive types and eliminators

$$\mathit{vcons} : \mathbf{\Pi} A : \mathtt{Set}.\, \mathbf{\Pi} n : \mathit{nat}.\, A \to \mathit{vec}\ A\ n \to \mathit{vec}\ A\ (S\ n)\}$$

$$\mathbf{Ind}_0\{\mathit{FTree} : \mathtt{Type}_0, \mathit{Forest} : \mathtt{Type}_0 := \mathit{leaf} : \mathit{FTree},$$

$$\mathit{node} : \mathit{Forest} \to \mathit{FTree}, \mathit{Fnil} : \mathit{Forest}, \mathit{Fcons} : \mathit{FTree} \to \mathit{Forest} \to \mathit{Forest}\}$$

Figure 4.2 shows the typing rules for inductive types and their eliminators. Rule `Ind-WF` describes when an inductive type is well-formed. Here, A_{d_i} is a sort that is called the arity of the inductive type d_i. This rule requires that all inductive types and constructors of the block are well-typed. The set $\mathsf{Constrs}(\Delta_C, d)$ is the set of constructors in Δ_C that produce something of type d. The proposition $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ describes the syntactic constraints for well-formedness of an inductive block. For precise details see Appendix A. It requires that all inductive types and all constructors of the block have as their first arguments the parameters of the block, e.g., A in *list* above. The parameters must be fixed for the whole block. In particular, the codomain type of each constructor must construct an inductive type that is applied to the parameters of the block, i.e., every constructor of *list* must construct a term of

type *list* A. All inductive types above satisfy these criteria. Both constructors of the type *vec*, for instance, start with the argument $A : \texttt{Type}_0$ and also they both construct a vector *vec* A n for some natural number n. Moreover, all arguments of constructors that are vectors take the same parameter A. This is the essential difference between parameters and indices. In addition, $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ also requires that all occurrences of inductive types of the block in any of the constructors of the block are strictly positive.

Remark 4.2.1. *Note that the names of inductive types and constructors of an inductive block in a typing context are* not *part of the domain of that context. We never refer to an inductive type or constructor without mentioning the block. We always write $\mathcal{D}.x$ for an inductive type or a constructor x in the block $\mathcal{D}$. In particular, we require for well-formed contexts that no variable appears in the domain of the context more than once. This restriction does not apply to inductive types as we can have multiple inductive types that share the same name for inductive types and/or constructors. This is a generalization of the global inductive declarations in the implementation of Coq.*

Eliminators In this work, we consider eliminators for inductive types as opposed to Coq's structurally recursive definitions, *i.e.*, `Fixpoint`s and `match` blocks in Coq. Note, however, that these can be encoded using eliminators as they are presented here (Paulin-Mohring, 1996) using the accessibility proof of the subterm relation, definable for any (non-propositional) inductive family.

Rule Ind-Elim in Figure 4.2 describes the typing for eliminators. Inductive types in a mutual inductive block can appear in one another. Hence, we define the elimination of inductive types for the entire block. We write $\textbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}$ for the elimination of t that is of type of the inductive type $\mathcal{D}.d_k$ (applied to values for parameters (these must be precisely $\vec{u}$) and indices). The term Q_{d_i} is the *motive* of elimination for the inductive type $\mathcal{D}.d_i$. This is basically a function that given the $\vec{a}$ and v such that v has type $\mathcal{D}.d_i\ \vec{u}\ \vec{a}$ produces a type (a term of some sort s'). The idea is that eliminating the term v should produce a term of type $Q_{d_i}\ \vec{u}\ \vec{a}\ u$. Note that the elimination $\textbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}$ is a term of type $Q_{d_k}\ \vec{u}\ \vec{b}\ t$ where t has type $d_k\ \vec{u}\ \vec{b}$.

In the elimination above the terms f_{c_i} are *case-eliminators*. The case-eliminator f_{c_i} is a function that describes the elimination of terms that are constructed using the constructor c_i. The term f_{c_i} is a function. It takes arguments of the constructor c_i together with the result of elimination of the (mutually) recursive arguments and produces a term of the appropriate type (according to the motives). This type is exactly what is formally defined as the type of the case eliminator for constructor c_i, $\xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i))$. The formal definition of the

$$\textsc{Beta}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma, x : A \vdash B : s \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.\, M)\ N \simeq M[N/x] : B[N/x]}$$

$$\textsc{Eta}\ \frac{\Gamma \vdash t : \mathbf{\Pi} x : A.\, B}{\Gamma \vdash t \simeq \lambda x : A.\, t\ x : \mathbf{\Pi} x : A.\, B}$$

Figure 4.3: An excerpt of judgemental equality rules

types of case-eliminators can be found in Appendix A . A simple example of eliminator is the induction principle for natural numbers:

$$\lambda P : nat \to \texttt{Prop}.\, \lambda pz : P\ Z.\, \lambda ps : \mathbf{\Pi} x : nat.\ P\ x \to P\ (S\ x).$$

$$\lambda n : nat.\, \mathbf{Elim}(n; nat; \mathsf{nil}; P)\ \{pz, ps\}$$

which has the type $\mathbf{\Pi} P : nat \to \texttt{Prop}.\, (P\ Z) \to (\mathbf{\Pi} x : nat.\, P\ x \to P\ (S\ x)) \to \mathbf{\Pi} n : nat.\, P\ n$.

Judgemental equality Figure 4.3 depicts an excerpt of the rules for judgemental equality. The rules Beta and Eta correspond to β and η equivalence. In this figure, we have elided the rules that specify that judgemental equality is an equivalence relation. The rules Delta, Zeta and Iota, respectively corresponding to unfolding definitions, expansion of let-ins and simplification of eliminators are also elided in Figure 4.3. The rule Iota basically states that when the term being eliminated is a constructor c applied to certain values, then the result of elimination is judgementally equal to the corresponding case-eliminator f_c applied to the arguments of the constructor where (mutually) recursive arguments are appropriately eliminated. See Appendix A for details. Note that the equivalence of the judgmental equality presentation and the implementation of definitional equality by conversion (as implemented in Coq) is a tricky issue and it is still an open problem to formally show equivalence for a system with cumulativity Siles and Herbelin, 2012, we leave this to future work.

Conversion/Cumulativity Figure 4.4 shows an excerpt of conversion/cumulativity rules. The core of these rules is the rule Cum. It states that whenever a term t has type A and the conversion/cumulativity relation $A \preceq B$ holds, then t also has type B. The rule Eq-Cum says that two judgementally equal (convertible) types M and M' are in conversion/cumulativity relation $M \preceq M'$.

$$\textsc{Prop-in-Type}\quad \Gamma \vdash \texttt{Prop} \preceq \texttt{Type}_i \qquad \textsc{Cum-Type}\quad \frac{i \leq j}{\Gamma \vdash \texttt{Type}_i \preceq \texttt{Type}_j}$$

$$\textsc{Cum-Prod}\quad \frac{\Gamma \vdash A_1 \simeq B_1 : s \qquad \Gamma, x : A_1 \vdash A_2 \preceq B_2}{\Gamma \vdash \mathbf{\Pi} x : A_1.\, A_2 \preceq \mathbf{\Pi} x : B_1.\, B_2} \qquad \textsc{Cum}\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash t : B}$$

$$\textsc{Eq-Cum}\quad \frac{\Gamma \vdash M \simeq M' : s}{\Gamma \vdash M \preceq M'}$$

Figure 4.4: An excerpt of conversion and cumulativity rules of pCIC

The rules Prop-in-Type and Cum-Type specify the order on the hierarchy of sorts. The rule Cum-Prod states the conditions for conversion/cumulativity between two (dependent) function types. Note that in this rule, Π-types are *not* contravariant w.r.t. their domain. This is also the case in Coq. This condition is crucial for the construction of our set-theoretic model, since set-theoretic functions (*i.e.*, functional relations) are not contravariant.

4.3 Universes in Coq and pCIC

In the system that we have presented in this section, and for most of this paper, universe levels, *e.g.*, i in $\texttt{Type}_i$, are explicitly specified. However, Coq enjoys a feature known as *typical ambiguity*. That is, users need not write universe levels explicitly; these are inferred by Coq. The idea here is that it suffices that there are universe levels, that can be placed in the appropriate places in the code, so that the code makes sense and respects consistent universe constraints. From a derivation with a consistent set of universe constraints one can always derive a pCIC derivation, using a valuation of the floating universe variables into the $\mathbb{U}_0 \ldots \mathbb{U}_n$ universes. This is exactly what is guaranteed using global universes and a global set of constraints on universe variables. In this sense the system pCIC as briefly discussed above forms a basis for Coq.

Universe polymorphism (Sozeau and Tabareau, 2014) extends Coq so that constructions can be made universe-polymorphic, *i.e.*, parameterized by some universe variables, following Harper and Pollack's seminal work (Harper and Pollack, 1991). That is, each universe-polymorphic definition will carry a context of universes together with a local set of constraints. The idea here is that any instantiation of a universe-polymorphic construction with universe levels that

satisfy the local constraints is an acceptable one. In the implementation of conversion, universe levels only play a role when comparing two sorts or two polymorphic constants, inductives or constructors. In the kernel of Coq, only checking of the constraints is involved, they are hence global to a whole term type-checking process. The system is justified by a translation to pCIC as well, making "virtual" copies of every instance of universe-polymorphic constants and inductive types.

In this section we discuss these two features and how they treat inductive definitions. For the rest of this paper we will consider the systems pCIC and its extension pCuIC without either typical ambiguity or universe polymorphism. When describing the system pCuIC we will consider how changes to the base theory allows a different treatment of universe-polymorphic inductive types compared to pCIC.

Typical ambiguity, global algebraic universes and template polymorphism The user can only specify `Prop`, `Set` or `Type`. This is done by considering a collection of global algebraic universes (as opposed to local ones in universe-polymorphic constructions as we will see). These universes are generated from the carrier set $\{\texttt{Set}\} \cup \{\mathbb{U}_\ell, |\ell \in \mathcal{L}\}$ for some countably infinite set of labels $\mathcal{L}$ (a.k.a. *levels*) with the operations max and successor $(+1)$ (constructing *algebraic* universes).[2] Each use of the sort `Type` is replaced with some $\texttt{Type}_{\mathbb{U}_\ell}$ for some fresh universe level ℓ. A global *consistent* set of constraints on the universe levels is kept at all times. When Coq type checks a construction, it may add some constraints to this set. If adding a constraint would render the constraints inconsistent then the definition at hand is rejected with a *universe inconsistency* error. Let us consider the example of lists in Coq[3].

```
Inductive list (A : Type@{𝕌ℓ}) : Type@{𝕌ℓ} :=
| nil : list A | cons : A → list A → list A.
  (* constraint added : 𝕌ℓ > Set *)
```

When Coq processes the inductive definition of lists above, one constraint about $\mathbb{U}_\ell$ is added to the set of constraints, enforcing $\texttt{Set} < \ell$, as ℓ is global. The following set of constraints are added with the following definitions:

```
Definition nat_list := list nat.
  (* constraint added : 𝕌ℓ ≥ Set, already implied *)
Definition Set_list := list Set.
  (* constraint added : 𝕌ℓ > Set, already implied *)
```

[2] In Coq, the sort `Prop` is treated in a special way. In particular, `Prop` is never unified with a universe $\texttt{Type}_{\mathbb{U}_\ell}$ for any algebraic universe $\mathbb{U}_\ell$.

[3] Here we show algebraic universes for the sake of clarity. These neither need to be written by the user nor are visible unless explicitly asked for. From now on, we will freely mention universe levels and constraints for presentation purposes but they can all be omitted.

```
Definition Type_list := list Type.
  (* constraint added : 𝕌_ℓ > 𝕌_ℓ' for
   some fresh 𝕌_ℓ' for the occurrence of Type *)
```

Template Polymorphism Template polymorphism is a simple form of universe polymorphism for *non-universe-polymorphic* inductive types. It only applies to inductive types whose sort contain levels that appear *only* in one of their parameters and nowhere else in that inductive type. A prime example is the definition of `list` above. The sort of the inductive type appears only in the type of the only parameter. In case template polymorphism applies, different instantiations of the inductive types with different arguments for parameters can have different types. For instance, the terms above have different types:

```
Check (list nat). (* list nat : Set *)
Check (list Set). (* list Set : Type@{Set+1} *)
```

Here `Type@{𝕌}` is Coq syntax for $\mathtt{Type}_{\mathbb{U}}$. This feature is very important for reusability of the basic constructions such as lists. Crucially, template polymorphism considers two instances of a template-polymorphic inductive type convertible, whenever they are applied to arguments that are convertible, regardless of the universe in which these arguments are considered. That is, the following Coq code type checks.

```
Universe i j. Constraint i < j.
Definition list_eq : list (nat : Type@{i}) = list (nat : Type@{j}) :=
  eq_refl.
```

Universe polymorphism in pCIC and inductive types The system pCIC has been extended with universe polymorphism (Sozeau and Tabareau, 2014). This allows for definitions to be parameterized by universe levels. The essential idea here is that instead of declaring global universes for every occurrence of `Type` in constructions, we use *local* universe levels (always $\geq$ `Set`, which we omit in local constraints). That is, each universe-polymorphic construction carries with itself a context of universe variables for universes that appear in the type and body of the construction together with a set of local universe constraints. These constraints may also mention global universe variables. This could happen in cases where the universe-polymorphic construction mentions universe-monomorphic constructions.

This feature allows us to define universe-polymorphic inductive types. The prime example of this is the polymorphic definition of categories:

```
Record Category@{i j} :=
  { Obj : Type@{i}; Hom : Obj → Obj → Type@{j}; ... }.
```

```
  (* local constraints: ∅ *)
```

This also allows us to define the category of (relatively small) categories as follows:[4]

```
Definition Cat@{i j k l} : Category@{i j} :=
  { Obj : Category@{k l}; ... }.
  (* local constr.: {k < i, l < i, k ≤ j, l ≤ j} *)
```

See Timany and Jacobs (2016a) for more details on using universe levels and constraints of Coq to represent (relative) smallness and largeness in category theory.

Note that the construction above, of the category of (relatively small) categories, could not be done in a similar way with a universe-monomorphic definition of category. This is because, the constraint `k` $<$ `i` would be translated to $\mathbb{U} < \mathbb{U}$ for some algebraic universe $\mathbb{U}$ that is taken to stand for the type of objects of categories. This would immediately make the global set of universe constraints inconsistent and thus the definition of category of categories would be rejected with a universe inconsistency error. Also notice that the universe-monomorphic version of the type `Category` is *not* template-polymorphic as the universe levels in the sort appear in the *constructor* of the type, and not only in its parameters and type.

Universe polymorphism treats inductive types at different universe levels as different types with no relation between them. This means that, in order to have a subtyping/cumulativity relation between two inductive types it requires the two instances to be at the exact same level. That is, for the subtyping relation `Category@{i j}` $\preceq$ `Category@{i' j'}` to hold it is required that `i` $=$ `i'` and `j` $=$ `j'`. This means, among other things, that the category of categories defined above is not the category of all categories that are at most as large as `k` and `l` but those categories that are exactly at the level `k` and `l`.

This is not only about small and large objects like categories. Let `A : Type@{i}` be a type, obviously, `A : Type@{j}`, for any `j` $>$ `i`. However, for the universe-polymorphic definition of lists, `uplist`, the types `uplist@{i} (A : Type@{i})` and `uplist@{j} (A : Type@{j})` are neither judgementally equal nor does the expected subtyping relation hold. In other words, the following Coq code will be accepted by Coq, *i.e.*, the reflexivity tactic will fail.[3]

```
Polymorphic Inductive uplist@{k} (A : Type@{k}) : Type@{k} :=
| upnil : uplist A | upcons : A → uplist A → uplist A.
```

[4]There can be some other local constraints that we have omitted given rise to by mixing of universe-polymorphic and universe-monomorphic constructions, *e.g.*, if the definition of categories or **Cat** uses some universe-monomorphic definitions from the standrad library of Coq.

```
Universe i j. Constraint i < j.
Lemma uplist_eq :
 uplist@{i} (nat : Type@{i}) = uplist@{j} (nat : Type@{j}).
  Fail reflexivity.
Abort.
```

As we discussed and demonstrated earlier, a similar equality with universe-monomorphic definition of lists does indeed hold. Note that the manually added constraint, `Constraint i < j`, is crucial here as otherwise the `reflexivity` tactic would succeed and Coq would silently equate universe levels `i` and `j`.

4.4 The predicative calculus of cumulative inductive constructions (pCuIC)

The system pCuIC extends the system pCIC by adding support for cumulativity between inductive types. This allows for different instances of a polymorphic inductive definition to be treated as subtypes of some other instances of the same inductive type under certain conditions.

The intuitive definition The intuitive idea for subtyping of inductive types is that an inductive type I is a subtype of another inductive type I' if they have the same *shape*, *i.e.*, the same number of parameters, indices and constructors and corresponding constructors take the same number of arguments. Furthermore, it should be the case that every corresponding index (note that these do not include parameters) and every corresponding argument of every corresponding constructor have the expected subtyping relation (the one from I is a subtype of the one from I', *i.e.*, covariance) and also that corresponding constructors have the same end result type. One crucial point here is that we *only* compare inductive types if they are fully applied, *i.e.*, there are values applied for every parameter and index. This is because the cumulativity relation is only defined for types and not general arities.

Put more succinctly, given a term of type I applied to parameters and indices, it can be destructed and then reconstructed using the corresponding constructor of I', *i.e.*, terms of type I can be lifted to terms of type I' using identity coercions. Note that we do not consider parameters of the inductive types in question. This is because parameters of inductive types are basically forming different families of inductive types. For instance, the type `list A` and `list B` are two different families of inductive types. Not considering parameters allows our cumulativity relation for universe-polymorphic inductive types to mimic the behavior of

template-polymorphic inductive types where the type of lists of a certain type are considered judgementally equal regardless of which universe level the type in question is considered to be in. Consider the following examples:

Example: categories The type `Category` is defined as a record. A record is an inductive type with a single constructor. In this case, there are no parameters or indices. The single constructors are constructing the same end result, *i.e.*, `Category`. As a result, in order to have the expected subtyping relation between `Category@{i j}` $\preceq$ `Category@{i' j'}`, `i` $\leq$ `i'` and `j` $\leq$ `j'`, we need to have that these constraints suffice to show that every argument of the constructor of `Category@{i j}` is a subtype of the corresponding argument of the constructor of `Category@{i' j'}`. Note that it is only the first two arguments of the constructors that differ between these two types. The rest of the arguments, *e.g.*, composition of morphisms, associativity of composition, etc., are identical in both types. Hence, we only need to have the subtyping relations [5] $\texttt{Type}_i \preceq \texttt{Type}_{i'}$ and $Obj \to Obj \to \texttt{Type}_j \preceq Obj \to Obj \to \texttt{Type}_{j'}$ to hold and they do hold.

Example: lists The type of lists has a single parameter and no index, also notice that the universe level `i` in `list@{i}` does not appear in any of the two constructors. Hence, the subtyping relation `list@{i} A` $\preceq$ `list@{j} A` holds for any type `A` regardless of the relation between `i` and `j`.

Figure 4.5 shows the typing rules for cumulativity of inductive types. The rule `C-Ind` describes the condition for subtyping of inductive types $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.d\ \vec{a}$. This subtyping relation holds if the two types are fully applied, that is, the applications are terms of some sort s and s' respectively. It is also required that the inductive blocks $\mathcal{D}$ and $\mathcal{D}'$ are related under the $\preceq^{\dagger}$ relation. The rule Ind-leq is rather lengthy but it essentially states what we explained above intuitively. It says that the relation $\mathcal{D} \preceq^{\dagger} \mathcal{D}'$ holds if the two blocks are defining inductive types with the same names and constructors with the same names. It also requires that for every corresponding inductive type in these blocks, the corresponding indices, are in the expected subtyping relation; similarly for corresponding arguments of corresponding constructors. Furthermore, corresponding constructors need to construct judgementally equal results.

Judgemental equality of inductive types Figure 4.6 shows the typing rules for judgemental equality of inductive types and their constructors. The rule

[5] For the sake of clarity we have omitted the context under which these cumulativity relations need to hold.

IND-LEQ

$$\frac{\begin{array}{c}\mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \qquad \mathcal{D}' \equiv \mathbf{Ind}_n \{\Delta'_I := \Delta'_C\} \in \Gamma \\ \mathsf{dom}(\Delta_I) = \mathsf{dom}(\Delta'_I) \qquad \mathsf{dom}(\Delta_C) = \mathsf{dom}(\Delta'_C) \\ \Big[\Delta_I(d) \equiv \vec{p} : \vec{P}.\mathbf{\Pi}\vec{z} : \vec{V}.\, s \qquad \Delta'_I(d) \equiv \vec{p} : \vec{P'}.\mathbf{\Pi}\vec{z} : \vec{V'}.\, s' \\ \Gamma, \vec{p} : \vec{P} \vdash \vec{V} \preceq \vec{V'} \\ \Big(\Delta_C(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P}.\mathbf{\Pi}\vec{x} : \vec{U}.\, d\ \vec{u} \qquad \Delta'_C(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P'}.\mathbf{\Pi}\vec{x} : \vec{U'}.\, d\ \vec{u'} \\ \Gamma, \vec{p} : \vec{P} \vdash \vec{U} \preceq \vec{U'} \qquad \Gamma, \vec{p} : \vec{P}, \vec{x} : \vec{U} \vdash \vec{u} \simeq \vec{u'} : \vec{P'}, \vec{V'} \\ \text{for } c \in \mathsf{Constrs}(\Delta_C, d)\Big) \qquad \text{for } d \in \mathsf{dom}(\Delta_I)\Big]\end{array}}{\Gamma \vdash \mathcal{D} \preceq^{\dagger} \mathcal{D}'}$$

C-IND

$$\frac{\mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \qquad \mathcal{D}' \equiv \mathbf{Ind}_n \{\Delta'_I := \Delta'_C\} \qquad \Gamma \vdash \mathcal{D} \preceq^{\dagger} \mathcal{D}' \qquad \Gamma \vdash \mathcal{D}.d\ \vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} : s'}{\Gamma \vdash \mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a}}$$

Figure 4.5: Cumulativity for inductive types

IND-EQ

$$\frac{\Gamma \vdash \mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} \preceq \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.d\ \vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} : s}{\Gamma \vdash \mathcal{D}.d\ \vec{a} \simeq \mathcal{D}'.d\ \vec{a} : s}$$

CONSTR-EQ-L

$$\frac{\Gamma \vdash \mathcal{D}'.d\ \vec{a} \preceq \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.c\ \vec{m} : \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}{\Gamma \vdash \mathcal{D}.c\ \vec{m} \simeq \mathcal{D}'.c\ \vec{m} : \mathcal{D}.d\ \vec{a}}$$

CONSTR-EQ-R

$$\frac{\Gamma \vdash \mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.c\ \vec{m} : \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}{\Gamma \vdash \mathcal{D}.c\ \vec{m} \simeq \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}$$

Figure 4.6: Judgemental equality for inductive types

Ind-Eq states that two inductive types are considered to be judgementally equal if they are in mutual cumulativity relations.

This, and the judgemental equality for constructors explained below, allow universe polymorphism to mimic the behavior of template polymorphism for monomorphic inductive types. For instance, as we saw types `list@{i} A` is a subtype of `list@{j} A` for any type `A` regardless of `i` and `j`. Hence, using the rule Ind-Eq it follows that the two types `list@{i} A` and `list@{j} A` are judgementally equal. However, the conditions of judgemental equality of universe-polymorphic inductive types is much more general compared to the conditions for template polymorphism to apply. Template polymorphism simply does not apply as soon as the universe in the sort is mentioned in any of the constructors.

According to the rule Ind-Eq, in order to get that the two types `Category@{i j}` and `Category@{i' j'}` are judgementally equal it is required that `i` = `i'` and `j` = `j'` as expected.

Judgemental equality of constructors The rules Constr-Eq-L and Constr-Eq-R specify judgemental equality of constructors of inductive types in cumulativity relation. Let $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.d\ \vec{a}$ be two inductive types in the cumulativity relation $\mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a}$. Furthermore, let c be a constructor of the inductive blocks $\mathcal{D}$ and $\mathcal{D}'$ and $\vec{m}$ be terms such that $\mathcal{D}.c\ \vec{m}$ has type $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.c\ \vec{m}$ has type $\mathcal{D}'.d\ \vec{a}$. In this case, the rules Constr-Eq-L and Constr-Eq-R specify that $\mathcal{D}.c\ \vec{m}$ and $\mathcal{D}'.c\ \vec{m}$ are judgementally equal *at the highest* of the two types $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.d\ \vec{a}$.

This is another behavior of template polymorphism that the rules Constr-Eq-L and Constr-Eq-R allow us to mimic. For instance, consider the monomorphic and template-polymorphic inductive type of lists defined above. Template polymorphism of list implies that, *e.g.*, the empty list (the constructor `nil`) for the type of lists of a type A are judgementally equal regardless of the sort that A is in. That is, we have `nil (A : Type@{i})` $\simeq$ `nil (A : Type@{j})` regardless of `i` and `j`. Using the rules Constr-Eq-L and Constr-Eq-R we can achieve a similar result for the universe-polymorphic and inductive type of lists `uplist` defined above. These rules imply that `upnil@{i} A` $\simeq$ `upnil@{j} A` for any type `A` regardless of `i` and `j`.

4.5 Consistency

We establish the consistency of pCuIC by constructing a set theoretic model for the theory inspired by the model constructed by Lee and Werner (2011).

$$\llbracket \Gamma \vdash \texttt{Prop} \rrbracket_\gamma \triangleq \{\emptyset, \{\emptyset\}\} \qquad \llbracket \Gamma \vdash \texttt{Type}_i \rrbracket_\gamma \triangleq \mathcal{V}_{\kappa_i}$$

$$\llbracket \Gamma \vdash t\ u \rrbracket_\gamma \triangleq \mathsf{App}(\llbracket \Gamma \vdash t \rrbracket_\gamma, \llbracket \Gamma \vdash u \rrbracket_\gamma)$$

$$\llbracket \Gamma \vdash \mathbf{\Pi} x : A.\, B \rrbracket_\gamma \triangleq \left\{ \mathsf{Lam}(f) \middle| f : \Pi a \in \llbracket \Gamma \vdash A \rrbracket_\gamma.\ \llbracket \Gamma, x : A \vdash B \rrbracket_{\gamma,a} \right\}$$

$$\llbracket \Gamma \vdash \lambda x : A.\, t \rrbracket_\gamma \triangleq \mathsf{Lam}\left(\left\{ (a, \llbracket \Gamma, x : A \vdash t \rrbracket_{\gamma,a}) \middle| a \in \llbracket \Gamma \vdash A \rrbracket_\gamma \right\} \right)$$

Figure 4.7: Excerpts of the model

We use our model to show (using relative consistency) that there are types that are not inhabited in the system. In fact, the model of Lee and Werner (2011) does support cumulativity of inductive types. However, it is not suitable for showing consistency as it relies on the normalization of the body of fixpoints (structural recursion in Coq) for interpreting them. Furthermore, we work in ZFC set theory and use the axiom of choice *only* to show that the interpretation of inductive types constructed through fixpoints does indeed belong to the interpretation of the sort of the inductive type. Lee and Werner (2011) work in ZF (with suitable cardinals, similarly to what we have assumed below) but we were not able to find a proof of this aspect of correctness of their interpretation of inductive types. See Appendix B for details of why and how we use the axiom of choice.

The model Here, we briefly present the most important parts of the model (see Appendix B for details of the construction). We construct our set theoretic model in ZFC set theory together with the axiom that there is a strictly increasing sequence of uncountable strongly inaccessible cardinals: $\kappa_0, \kappa_1, \ldots$ with $\kappa_0 > \omega$. Universe $\texttt{Type}_i$ is interpreted as set theoretic (von Neumann) universes $\mathcal{V}_{\kappa_i}$ Drake, 1974. It is well-known Drake, 1974 that the von Neumann universe $\mathcal{V}_\kappa$ is a model of ZFC for any uncountable strong inaccessible cardinal κ. We interpret the sort Prop as the set $\{\emptyset, \{\emptyset\}\}$. Figure 4.7 shows excerpts of our model of pCuIC. Interpretation of inductive types and eliminators are discussed below. We write $A\downarrow$ for well-definedness of the object A. We write $\Pi a \in A.\, B(a)$ for dependent set theoretic functions: $\Pi a \in A.\, B(a) \triangleq \left\{ f \in \left(\bigcup_{a \in A} B(a)\right)^A \middle| \forall a \in A.\ f(a) \in B(a) \right\}$. Here Lam and App are respectively functions that trace-encode a set-theoretic function and evaluate a trace encoded functions. Trace encoding is a standard technique Aczel, 1999 for set-theoretic representation of functions in a type theory with a proof-irrelevant universe (Prop in our case) which is a sub-type of another non-proof-irrelevant universe ($\texttt{Prop} \preceq \texttt{Type}_i$ in our case).

Modeling inductive types and eliminators The basic idea of the interpretation of inductive types, constructors and eliminators is straightforward. However, the general presentation of the construction is lengthy and involves arguments regarding the general shape of inductive types. In particular, the strict positivity condition plays a crucial role. Here, we present the general idea and give some examples. Further details are available in Appendix B . Following Lee and Werner (2011), who follow Dybjer (1991) and Aczel (1999), we use inductive definitions (in set theory) constructed through rule sets to model inductive types. Here, we give a very short account of *rule sets* for inductive definitions. For further details refer to Aczel (1977). A rule set is a set of rules. A pair (A, a) is a rule based on a set U where $A \subseteq U$ is the set of premises and $a \in U$ is the conclusion. We write $\frac{A}{a}$ for a rule (A, a). The fixpoint $\mathcal{I}(\Phi)$ of a rule set Φ is the smallest set X such that for any rule $\frac{A}{a}$ if $A \subseteq X$ then $a \in X$. Every rule set has a fixpoint (Aczel, 1977).

The idea here is to construct a rule set for the whole inductive block. For each collection of arguments that can possibly be applied to a constructor we add a rule to the rule set. The premises of the rule requires that all (mutually) recursive arguments are in the fixpoint. We define the interpretation of individual inductive types based on this fixpoint. Let $\mathcal{D} \equiv \mathbf{Ind}_0\{nat : \texttt{Set} := Z : nat, S : nat \to nat\}$ be the inductive block for inductive definition of natural numbers. The rule set for this inductive block is as follows:

$$\Phi_{\mathcal{D}} \triangleq \left\{ \frac{\emptyset}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 0; \mathsf{nil} \rangle \rangle} \right\} \cup \left\{ \frac{\{\langle 0; \mathsf{nil}; \mathsf{nil}; a \rangle\}}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 1; a \rangle \rangle} \middle| a \in \mathcal{V}\kappa_0 \right\}$$

The rule corresponding to Z has no premise as Z takes no recursive argument. This rule concludes that the term $\langle 0; \mathsf{nil} \rangle$, *i.e.*, zero$^{\text{th}}$ constructor applied to nil arguments is a term of zero$^{\text{th}}$ type with nil as both parameters and indices. The rules corresponding to S state that $\langle 1; a \rangle$ is an element of the zero$^{\text{th}}$ type if a is. Based on this fixpoint we define the semantics of natural numbers, $[\![\cdot \vdash \mathcal{D}.nat]\!]_{\mathsf{nil}} \triangleq \{\langle k; \vec{a} \rangle | \langle 0; \mathsf{nil}; \mathsf{nil}; \langle k; \vec{a} \rangle \rangle \in \mathcal{I}(\Phi_{\mathcal{D}})\}$, zero, $[\![\cdot \vdash \mathcal{D}.Z]\!]_{\mathsf{nil}} \triangleq \langle 0; \mathsf{nil} \rangle$ and successor, $[\![\cdot \vdash \mathcal{D}.S]\!]_{\mathsf{nil}} \triangleq \mathsf{Lam}\,(\{(a, \langle 1; a \rangle) | a \in [\![\cdot \vdash \mathcal{D}.nat]\!]_{\mathsf{nil}}\})$.

Interpreting eliminators We use rule sets to also define the interpretation of eliminators. For each constructor applied to a sequence of arguments we add a rule to the rule set. This rule states that the result of elimination is exactly the result of applying the corresponding case eliminator where the result of elimination of (mutually) recursive arguments are taken as arbitrary sets. The premise requires that each set, that is taken as elimination of a (mutually) recursive argument, is mapped correctly in the fixpoint. We define the interpretation of elimination of a term t of an inductive type as the set a if $[\![t]\!] = \langle k; \vec{m} \rangle$ and a is the unique set such that the pair $(\langle k; \vec{u}, \vec{m} \rangle, a)$ is in

the fixpoint of the elimination, where $\vec{u}$ is the interpretation of the values for parameters. Assume we have sets r, rz and rs such that $r, rz, rs \in [\![\Gamma]\!]$ where $\Gamma = Q : nat \to \mathtt{Type}_i, qz : Q\ Z, qs : \mathbf{\Pi} x : nat.\, Q\ x \to Q\ (S\ x)$. The rule set for the elimination of natural numbers is as follows:

$$\Phi_{ELB} \triangleq \left\{ \frac{\emptyset}{(\langle 0; \mathsf{nil} \rangle, rz)} \right\} \cup \left\{ \frac{\{(a,b)\}}{(\langle 1; a \rangle, \overrightarrow{\mathsf{App}}(rs, a, b))} \,\middle|\, \begin{array}{l} a \in [\![\Gamma \vdash \mathcal{D}.nat]\!]_{r,rz,rs}, \\ b \in \mathcal{V}\kappa_i \end{array} \right\}$$

We define the interpretation of elimination of the term n as a if $[\![\Gamma \vdash n]\!]_{r,rz,rs} = \langle k, \vec{m} \rangle$ and a is the unique set such that the pair $(\langle k; \vec{m} \rangle, a) \in \mathcal{I}(\Phi_{ELB})$. Notice, that here we have no parameters and thus we have not added the interpretation of values for parameters in the tuple $\langle k; \vec{m} \rangle$.

Soundness theorem and consistency

Theorem 4.5.1 (Soundness of the model)**.** *1. If $\mathcal{WF}(\Gamma)$ then $[\![\Gamma]\!]\downarrow$*

2. *If $\Gamma \vdash t : A$ then $[\![\Gamma]\!]\downarrow$ and for any $\gamma \in [\![\Gamma]\!]$ we have $[\![\Gamma \vdash t]\!]_\gamma\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\downarrow$ and $[\![\Gamma \vdash t]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$*
3. *If $\Gamma \vdash t \simeq t' : A$ then $[\![\Gamma \vdash t]\!]_\gamma\downarrow$, $[\![\Gamma \vdash t']\!]_\gamma\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\downarrow$ and $[\![\Gamma \vdash t]\!]_\gamma = [\![\Gamma \vdash t']\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$*
4. *If $\Gamma \vdash A \preceq B$ then $[\![\Gamma \vdash A]\!]_\gamma\downarrow$, $[\![\Gamma \vdash B]\!]_\gamma\downarrow$ and $[\![\Gamma \vdash A]\!]_\gamma \subseteq [\![\Gamma \vdash B]\!]_\gamma$*

Proof. By mutual induction on the typing derivations. For C-Ind we need to show that the interpretation of one inductive type is a subset of the interpretation of the other one. This follows from the fact that the arguments of constructors of the two types have the required subset relation (by induction hypothesis). The cases Ind-Eq, Constr-Eq-L and Constr-Eq-R are trivial. □

Corollary 4.5.2 (Consistency of pCuIC)**.** *Let s be a sort, then, there* does not *exist any term t such that $\cdot \vdash t : \mathbf{\Pi} x : s.\, x$.*

Proof. If there were such a term t, by Theorem 4.5.1 we would have $[\![\cdot \vdash t]\!]_{\mathsf{nil}} \in [\![\cdot \vdash \mathbf{\Pi} x : s.\, x]\!]_{\mathsf{nil}}$. However, $[\![\cdot \vdash \mathbf{\Pi} x : s.\, x]\!]_{\mathsf{nil}} = \emptyset$. □

4.6 Coq implementation

We implemented the extension to pCIC, that are presented in this paper, in the Coq system, which is now available as of the stable 8.7 version of the system

The Coq Development Team (2017), documented[6] and even experimented with already in the UniMath library.[7]

From the user point of view, this adds a new optional flag on universe-polymorphic inductive types that computes the cumulativity relation for two arbitrary fresh instances of the inductive type that can be printed afterwards using the `Print` command. Cumulativity and conversion for the fully applied inductive type and its constructors is therefore modified to use the cumulativity constraints instead of forcing equalities everywhere as was done before, during unification, typechecking and conversion. As cumulativity is always potentially more relaxed than conversion, users can set this option in existing developments and maintain compatibility. Of course actually making use of the new feature is not backward-compatible.

Impact on the Coq codebase The impact of this extension is relatively small as it involves mainly an extension of the data-structures representing the universes associated with polymorphic inductive types in the Coq kernel, and their use during the conversion test of Coq, which was already generic in the tests used for comparing polymorphic inductives and constructors. Note that we have not needed to adapt the two efficient *reduction* strategies of Coq, `vm_compute` and `native_compute`, as universes are irrelevant for reduction. A good chunk of changes involved cleanups of the kernel API for registering inductive declarations.

Performance When no inductive type is declared cumulative, the extension has no impact, as we tested on a large set of user contributions including the Mathematical Components and the Coq HoTT library (the common stress-tests for universes). When activated globally, we hit one case in the test-suite of Coq taken from the HoTT library where the computation of the subtyping relation for a given inductive blows up, due to conversion unfolding definitions to infer the subtyping constraints. In this case we know that the relation would be trivial (cumulativity collapses to equality), hence we were motivated to make the `Cumulative` flag optional. The performance is otherwise not affected, as far as we know.

[6] https://coq.inria.fr/refman/addendum/universe-polymorphism.html
[7] See the discussion on GitHub: https://github.com/UniMath/UniMath/issues/648

4.7 Applications

In this section we briefly discuss two motivating applications that are made possible thanks to the new cumulativity feature for inductive types that we have presented here.

Yoneda embedding Each category `C : Category@{i j}` is equipped with a *hom*-functor, `Hom_func : C × C^op → Type_Cat@{j}`. Here `Type_Cat` is the category of types and functions, which plays the role of the *Set* category. It is expected that one could define the Yoneda embedding `Y(C)` as `Curry Hom_func` where `Curry` is the exponential transpose of the cartesian closed structure of the category of categories `Cat`. However, the cartesian closed version of `Cat@{i' j' k' l'}` has the constraints $\mathtt{k}' = \mathtt{l}' = \mathtt{j}'$ and `Type_Cat@{j} : Category@{k j}` with the side constraint $\mathtt{j} < \mathtt{k}$. This means that `Type_Cat` is not an object of any cartesian closed version of `Cat` making it impossible to use `Curry` on `Hom_func`. See Timany and Jacobs (2016a) for a detailed discussion of this issue.

Cumulativity of inductive types solves this issue. In pCuIC, `Type_Cat` is indeed an object of a cartesian closed version of `Cat` at some higher universe level allowing us to directly use exponential transpose to define the Yoneda embedding.

Syntactical models of type theories In Boulier, Pédrot, and Tabareau (2017), Boulier *et al.* advocate the study of syntactical models of type theory, that is models defined by definitional translations from a source type theory to a target type theory. A definitional translation of dependent type theory must preserve its conversion relation, which is known as "computational soundness" in proof theory in general. In pCIC and pCuIC, it must preserve the cumulativity relation.

A most basic example of syntactical model is the "cross-bool" model, which interprets every type as the type itself crossed with booleans, *i.e.*, using a polymorphic pair type:

$$[\mathtt{Type}_i] = (\mathtt{Type}_i \times_{j,\mathtt{Set}} \mathbb{B}, \mathtt{true}) \text{ where } i < j \qquad [\![A]\!] = [A].1$$

Likewise, every term is interpreted as the term itself *plus a boolean*. This model can be used to show that type extensionality, hence univalence, is independent from CC_ω (*op. cit.*). However, this model does not scale to Coq's type theory as the cumulativity rule is not validated through the translation. Indeed to validate cumulativity one must have, assuming $i \leq k \wedge i < j \wedge k < l$:

$$[\![\mathtt{Type}_i]\!] \leq [\![\mathtt{Type}_k]\!] \triangleq (\mathtt{Type}_i \times_{j,\mathtt{Set}} \mathbb{B}, \mathtt{true}).1 \leq (\mathtt{Type}_k \times_{l,\mathtt{Set}} \mathbb{B}, \mathtt{true}).1$$

This judgement holds only if $j = l$ and $i = k$ in pCIC, and is relaxed to only $i = k$ in pCuIC. The latter constraint is forced due to the appearance of the types as parameters of the pair type. We can go one step further and define a specialized inductive type:

```
Inductive TyInterp@{i j | i < j} : Type@{j} := { T : Type@{i}; b : bool }.
```

The subtyping constraints on `TyInterp` will only require that $i \leq k$, as assumed! Note also that template polymorphism would not help here as the type is not a parameter anymore.

4.8 Future and related work

Moving from template polymorphism to universe polymorphism One motivation for this extension is the ability to explain away the so-called "template-polymorphic" inductive types of Coq in terms of cumulative universe-polymorphic inductive types, to put the system on clean and solid theoretical ground and finally switch the standard library of Coq to full universe polymorphism. Making the universe-monomorphic code using template-polymorphic inductives in the standard library interact with universe-polymorphic code is prone to introduce universe inconsistencies, the two systems working in quite different ways. Hence we have tried to set universe polymorphism on everywhere.

Our experiments are encouraging but not without issues. We are able to make the basic inductive types of the standard library cumulative universe-polymorphic, and all constants polymorphic (except in a few files devoted to the formalization of paradoxes). We found that the relaxed rule on constructors was necessary in some cases, this is a case where practice met theory: our model construction justified the required relaxation for these examples.

However, we hit an orthogonal problem with the definitions of modules and module types, used to formalize the number and finite map and set libraries for example, where definitions drastically change meaning when interpreted in universe-polymorphic mode. Indeed when a module parameter `A : Type` is declared in monomorphic mode, one gets a floating universe, *i.e.*, it is elaborated to $A : \texttt{Type}_\ell$ for some global universe ℓ. In univ. poly. mode it is elaborated to $A@\{\ell\} : \texttt{Type}_\ell$ instead, which can only be instantiated by `Prop` and types in `Set`, at the bottom of the hierarchy. The only way to fix this is to add user annotations in the files to switch between monomorphic and polymorphic mode, which is work-in-progress.

Strong normalization We believe that our extension to pCIC maintains strong normalization and that the model constructed by Barras (2012) could be easily extended to support our added rules.

Related Work We are not aware of any other system providing cumulativity on inductive types, neither Matita nor Lean, the closest cousins of Coq, implement cumulativity. They prefer the algebraic presentation of universes that is also used in Agda and where explicit lifting functions must be defined between different instances of polymorphic inductive types. In McBride (2015), McBride presents a proposal for internalizing "shifting" of universe-polymorphic constructions to higher universe levels akin to an explicit version of cumulativity that was further studied by Rouhling (2014), but parameterized inductive types are not considered there.

4.9 Conclusion

We have presented a sound extension of the predicative calculus of inductive constructions with cumulative inductive types, which allows to equip cumulative universe-polymorphic inductive types with definitional equalities and reasoning principles that are closer to the "informal" mathematical practice. Our system is implemented in the Coq proof assistant and is justified by a model construction in ZFC set theory. We hope to make this feature more useful and applicable once we resolve the remaining, orthogonal issue with the module system, allowing users of the standard library of Coq to profit from it as well.

Acknowledgements We thank the anonymous reviewers for their very useful comments. This work was partially supported by the CoqHoTT ERC Grant 637339 and partially by the Flemish Research Fund grants G.0058.13 and G.0962.17N.

Part II

Logical Relations: a Semantic Approach to the Study of Programming Languages Through Their Types

Table of symbols (Part II)

Programming languages studied

n	Natural number
$\circledcirc$	A general binary operation
τ	A type
Γ	A typing context (mapping variables to their types)
Ξ	A type-level context (a list of type variables)
$\Xi \mid \Gamma e \tau \vdash :$	Typing judgement
X	A type variable
$()$	The unit value
1	The unit type
$\mathbb{N}$	Type of natural number
$\mathbb{B}$	Type of Booleans
$\mathsf{EqType}(\tau)$	A type that `CAS` can be performed on
$\forall X.\, \tau$	A polymorphic type
$e\ \_$	Specialization of polymorphic term e
$\mu X.\, \tau$	A recursive type
$\texttt{ref}(\tau)$	A reference type
$\texttt{cont}(\tau)$	A continuation type
ℓ	A memory location
$\texttt{rec}\, f(x)() = e$	A recursive function f with body e
`fold`	Folding of recursive types

`unfold`	Unfolding of recursive types
`call/cc`	Call with current continuation
`throw`	Continue from a captured continuation
`CAS`	Atomic compare and set operation
$==$	Comparison of references (STLang)
$e[v/x]$	Substitution
$e[\vec{v}/\vec{x}]$	Simultaneous substitution
σ, h	State of the programming language, i.e., heap
$\uplus$	Disjoint union
$\rightarrow_{\mathsf{h}}$	Head step
$\rightarrow_K$	Head step under a specific evaluation context K ($\mathsf{F}^{\mu,ref}_{conc,cc}$)
$\rightarrow_{\mathsf{tp}}$	Thread pool step
$\rightsquigarrow$	Effectful reduction step (STLang)
$\rightarrow$	Reduction step (STLang)
$\rightarrow_d$	Deterministic reduction step (STLang)
$\rightsquigarrow_d$	Deterministic effecful reduction step (STLang)
K	Evaluation context
$\mathbb{K}$	Effectful evaluation context
$[], -$	The empty evaluation context
$e\downarrow, e\Downarrow$	e terminates
C	A well-typed context
$C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')$	Typing of contexts
$\Xi \mid \Gamma \vDash e \leq_{ctx} e' : \tau$	Contextual refinement
$\Xi \mid \Gamma \vDash e \approx_{ctx} e' : \tau$	Contextual equivalence

Iris

$\xrightarrow{\text{fin}}$	Partial function with finite support
$\mathcal{E}$	A mask
P, Q	An Iris proposition
Φ, Ψ	An Iris predicate

Expr	Type of expressions
Val	Type of values
Ectx	Type of evaluation contexts
$\top$	The true proposition; full mask (no invariant open)
$\top$	The false proposition; empty mask (all invariants open)
$*$	Separating conjunction
$-\!\!*$	The wand connective
$\Rightarrow$	Logical implication
$\triangleright$	The later modality
$\mu r.P$	Guarded fixpoint
$\Box$	The persistence modality
$\boxed{P}^{\mathcal{N}}$	An invariant with name $\mathcal{N}$
$\Rightarrow, \Rightarrow_{\mathcal{E}}, {}^{\mathcal{E}}\Rightarrow^{\mathcal{E}}$	The update modality
$\Rrightarrow$	Shorthand for $-\!\!* \Rightarrow$
$\mathsf{wp}\, e\, \{\Phi\}$	A weakest precondition proposition
$\{P\}\, e\, \{\Phi\}$	A Hoare triple
$\ell \mapsto v, \ell \mapsto_i v$	A maps-to proposition (memory location ℓ in the heap has value v)
$\ell \mapsto_s v$	A Specification side maps-to proposition (memory location ℓ in the heap of the specification side has value v)
$P \vdash Q$	Iris entailment
$P \dashv\vdash Q$	Logical equivalence of Iris propositions
$\mathsf{persistent}(P)$	P is persistent
$\mathsf{persistent}(\Phi)$	Φ is persistent, i.e., $\forall x.\ \mathsf{persistent}(\Phi(x))$

Logical relations

Δ	Interpretation of type variables (maps free type variables to their interpretation)
$[\![\Xi \vdash \tau]\!]_\Delta$	The value relation (unary or binary)

$[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta}$	The expression relation (unary or binary)
$[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}$	The typing contexts relation
$\Xi \mid \Gamma \vDash e : \tau$	The unary logical relation
$\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau$	The binary logical relation, also referred to as logical refinement
$\mathcal{K}[\![\Xi \vdash \tau]\!]_{\Delta}(K, K')$	The evaluation context relation
$\mathcal{O}(e, e')$	Observational refinement
$j \Mapsto K[e], j \Mapsto e$	Thread j is about to execute e (under the evaluation context K)

Iris propositions for stack refinement (Section 5.7)

$\mathrm{AllCells}(f)$	All stack cells are described by finite partial function f
$\ell \mapsto^{stk} v$	Stack cell ℓ stores value v
$\mathrm{LRel}_{\Phi}(\ell, v)$	The fine-grained stack (linked list) starting in ℓ is related to the coarse-grained stack (algebraic list) v and stored values are related by Φ

Iris propositions for the logical relation for monadic encapsulation of state (Chapter 6)

$\mathcal{E}\,[\![\Xi \vdash \tau]\!]_{\Delta}$	The expression relation
$\Rrightarrow\!\!\!\!\Mapsto$	The future modality
$\mathsf{IC}^{\gamma}\, e\, \{\!\{v.\ P\}\!\}$	If-Convergent (IC) predicate
$heap_{\gamma}(h)*$	The heap named γ has contents h
$\ell \mapsto_{\gamma} v$	The location ℓ has value v in the heap named γ
$(h, e) \Downarrow^{\gamma}_{\Phi}$	Expression e starting in a heap h reduces (deterministically) to a value satisfying Φ and a heap that is named by γ

Iris propositions for the logical relation for continuations (Chapter 7)

$\mathsf{clwp}\, e\, \{\Phi\}$	A context-local weakest precondition

Chapter 5

Type Soundness and Contextual Refinement via Logical Relations in Higher-Order Concurrent Separation Logic

It is well-known that it is challenging to construct semantic models for reasoning about type soundness and contextual refinement for programming languages with sophisticated type systems with polymorphic types, recursive types, and reference types. In this paper we present new semantic models based on logical relations for showing such properties. We present two logical relations for a programming language $\mathsf{F}_{\mu,ref,conc}$ featuring impredicative polymorphism, recursive types, dynamically allocated higher-order references and fine-grained concurrency. The first unary logical relation can be used to establish type safety of $\mathsf{F}_{\mu,ref,conc}$ and the second binary logical relation is useful for proving contextual refinements of programs. We define our logical relations in Iris, a state-of-the-art impredicative higher-order concurrent separation logic. The benefits of formalizing logical relations on top of Iris are two-fold: (1) it makes is simpler to define and reason about the logical relations; and (2) we can use the implementation of Iris in the Coq proof assistant to formalize our logical relations in the Coq proof assistant. We illlustrate the usefulness of our approach by proving, using the binary logical relation, that a fine-grained concurrent stack module refines a coarse-grained one. All of our results have been mechanized in the Coq proof assistant. Our development is the first formalization of logical relations for such a rich programming language in a proof assistant.

5.1 Introduction

Type soundness, introduced by Milner (1978), is one of the fundamental properties of a type system for a programming language. Type soundness expresses that well-typed programs are guaranteed to have well-defined behavior. Estalishing that a given type system guarantees type soundness is not a trivial problem. Initially, this problem was studied using denotational semantics (Damas, 1984; Milner, 1978) for the programming language in question. It turned out to be difficult to scale this methodology to more complex type systems and programming languages with features such as polymorphism and higher-order references, probably because denotational semantics was not sufficiently developed for such programming languages. This led to the development of so-called syntactic approaches to type soundness (Wright and Felleisen, 1994), which are based on the small-step operational semantics of the programming language. Harper (1994) formulated the approach in terms of progress and preservation lemmas, where progress expresses that a well-typed program is either a value or it can be evaluated further, and preservation expresses that a well-typed program remains well-typed when executed. This syntactic approach to type soundness scales very well to programming languages with sophisticated features and type systems. However, the resulting type soundness theorem is not as strong as we would really like: The syntactic type soundness theorem ignores

data abstraction aspects of the programming language – one can establish progress and preservation even if the language includes features that break data abstraction. Moreover, since it does not give a semantic characterization of type soundness it is not sufficiently modular: it only guarantees safety for syntactically well-typed programs. This severely limits the applicability of the approach since almost all realistic programming languages include some facility for mixing syntactially well-typed programs with syntactically ill-typed programs (which one then needs to guarantee are safe by other means than syntactic well typing).

In this paper we present a *semantic* approach to type soundness using logical relations based on the operational semantics of the programming language, sometimes known as operationally-based logical relations (Pitts, 1996). An advantage of using the logical relations approach is that it is both modular (since it gives a semantic characterization of type soundness) and also accounts for data abstraction. However, developing operationally-based logical relations for programming languages with advanced features such as higher-order references is non-trivial because of the so-called type-world circularity problem (Ahmed, 2004), which led to the development of so-called step-indexed Kripke logical relations with recursively defined worlds, see, e.g., Ahmed (2004), Birkedal, Støvring, and Thamsborg (2009), and Dreyer, Neis, and Birkedal (2010). Step-indexed Kripke logical relations are intricate and quite difficult to use for reasoning about examples. Here, instead, we take an alternative approach and use the Iris program logic to reason about these intricate parts of the model at a high level of abstraction. Iris is a state-of-the-art program logic based on separation logic designed for reasoning about concurrent higher-order imperative programs. The Iris program logic comes equipped with logical connectives and features that are sufficient for a direct definition of our logical relations. This allows us to define our logical relations at a higher level of abstraction and avoid step-indices and Kripke world details. We also use Iris when reasoning about logical relatedness of examples and thus avoid low-level reasoning about step-indices, etc.

Specifically, we present a unary logical relation for $\mathsf{F}_{\mu,ref,conc}$ in Iris. $\mathsf{F}_{\mu,ref,conc}$ is a call-by-value programming language featuring impredicative polymorphism (à la System F), recursive types, dynamically allocated higher-order references, and fine-grained concurrency. Fine-grained concurrent programs do not use locks for synchronization but rather use fine-grained concurrency primitives such as the atomic compare-and-set, `CAS`, operation which is a primitive of $\mathsf{F}_{\mu,ref,conc}$. We use the logical relation to show type soundness for syntactically well-typed programs and also demonstrate that the logical relation can be used to show semantic type soundness for a program that is not syntactically well-typed, thus making it possible to combine this program with syntactically well-typed

programs in a safe manner.

Our logical relation for type soundness is a unary logical relation. We also define binary logical relations for $\mathsf{F}_{\mu,ref,conc}$ in Iris and, moreover, we demonstrate that our binary logical relation is useful for showing challenging contextual refinements. In particular, we show that a fine grained implementation of a concurrent stack module refines a coarse grained implementation. The refinement is proved in Coq using the binary logical relation defined in Iris in Coq.

Our development is the first formalization of logical relations for such a rich programming language in a proof assistant.

Contributions In summary, the technical results presented in this paper are as follows:

- We formalize a unary logical relations model for $\mathsf{F}_{\mu,ref,conc}$ in Iris for proving type safety.
- We formalize a binary logical relations model for $\mathsf{F}_{\mu,ref,conc}$ in Iris for proving contextual refinements and equivalences.
- We use our binary logical relations model to prove contextual refinement for a pair of fine- and coarse-grained concurrent counter modules and a pair of fine- and coarse-grained concurrent stack modules.
- All results in this paper are mechanized in the Coq proof assistant, using the mechanization of Iris and the Iris Proof Mode (Krebbers, Timany, and Birkedal, 2017).

In addition to the technical contributions enumerated above, we also intend that this text may serve as a first introduction to the methodology of defining logical relations models in Iris. We believe that such logical relations models can be a powerful tool to tackle a wide gamut of applications in the theory of programming languages. In the present text we elucidate the technique for a fairly standard type system – we hope that understanding the models presented here should make it easier to understand more sophisticated models, such as Jung, Jourdan, Krebbers, and Dreyer, 2017; Krogh-Jespersen, Svendsen, and Birkedal, 2017; Timany, Stefanesco, Krogh-Jespersen, and Birkedal, 2018, and as-yet-to-appear models.

The technical results presented in this paper have previously been announced in a conference paper by Krebbers, Timany, and Birkedal (2017). In *loc. cit.*,

the results were only mentioned very briefly, as an application of the Iris proof mode.

Structure of the rest of the paper In Section 5.2 we present the syntax and semantics of $\mathsf{F}_{\mu,ref,conc}$. Section 5.3 gives a brief introduction to Iris. In Section 5.4 we present our unary logical relations model and use it to prove type soundness of $\mathsf{F}_{\mu,ref,conc}$. We present our binary logical relation model in Section 5.5 and show how it can be used to prove contextual refinements. We exemplify the use of our binary logical relations model in Sections 5.6 and 5.7. In these sections we sketch proofs of refinements of concurrent counter and stack modules, respectively. In Section 5.8 we discuss related work and conclude in Section 5.9.

5.2 The language $\mathsf{F}_{\mu,ref,conc}$: syntax and semantics

In this section we present the syntax and the semantics of our subject of study: $\mathsf{F}_{\mu,ref,conc}$. The programming language $\mathsf{F}_{\mu,ref,conc}$ is a call-by-value programming language with impredicative polymorphism, recursive types, dynamically allocated higher-order heap and fine-grained concurrency. The expressions, values and types of $\mathsf{F}_{\mu,ref,conc}$ are as follows:

$$e ::= x \mid () \mid n \mid e \circledcirc e \mid \texttt{rec}\, f(x) = e \mid e\ e \mid \Lambda\, e \mid e\ \_ \mid (e,e) \mid \pi_i\, e \mid \texttt{true} \mid$$
$$\texttt{false} \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{inj}_i\, e \mid \texttt{match}\, e\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} \mid$$
$$\texttt{fold}\, e \mid \texttt{unfold}\, e \mid \ell \mid \texttt{ref}(e) \mid\ !e \mid e \leftarrow e \mid \texttt{CAS}(e,e,e) \mid \texttt{fork}\ \{e\}$$

$$v ::= () \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid (v,v) \mid \texttt{inj}_i\, v \mid \texttt{fold}\, v \mid \ell$$

$$\tau ::= X \mid 1 \mid \mathbb{N} \mid \mathbb{B} \mid \tau \to \tau \mid \forall X.\, \tau \mid \tau \times \tau \mid \tau + \tau \mid \mu X.\, \tau \mid \texttt{ref}(\tau)$$

Here, x ranges over a countably infinite set of variables. The value () is the only inhabitant of the unit type. $\mathsf{F}_{\mu,ref,conc}$ includes natural numbers in its expressions and values. The symbol $\circledcirc$ ranges over binary operations (both arithmetic and comparison operations) on natural numbers. We omit details of syntax and semantics of these operations as these are entirely standard. We write $\texttt{rec}\, f(x) = e$ for a recursive function with argument x and body e. The expression $\Lambda\, e$ is the type-level lambda abstraction. The language $\mathsf{F}_{\mu,ref,conc}$ also features product types and sum types with projections and injections written as π_i and $\texttt{inj}_i$ respectively. As usual terms of the sum type can be eliminated

using a `match` statement which we write similarly to that of ML. The `fold` and `unfold` expressions respectively fold and unfold expressions of recursive types. Expressions and values of $\mathsf{F}_{\mu,ref,conc}$ also include memory locations ℓ. These can be allocated, read and written to using the $\texttt{ref}(e)$, $!\,e$ and $e \leftarrow e'$ expressions. The compare-and-set, `CAS`, expression is the $\mathsf{F}_{\mu,ref,conc}$ primitive for fine-grained concurrency. The expression $\texttt{CAS}(\ell, v, w)$ *atomically* checks if the value stored in memory at location ℓ is equal to v and, if so, sets the value of ℓ to w. The expression $\texttt{fork}\ \{e\}$ forks a new thread which starts evaluation of the expression e.

The syntax of $\mathsf{F}_{\mu,ref,conc}$ defined above is in the Curry style: types never appear in terms. This allows us to define the type system and our logical relations models over untyped terms. This enables us to reason about codes where well-typed code is mixed with untyped code (see Section 5.4 for more details).

The types of $\mathsf{F}_{\mu,ref,conc}$ include the basic types: the unit type, 1, the type of natural numbers, $\mathbb{N}$, and the type of booleans, $\mathbb{B}$. Basic type formers include function types, $\tau \to \tau'$, products, $\tau \times \tau'$ and sums $\tau + \tau$. Types also include polymorphic types, $\forall X.\, \tau$, and recursive types, $\mu X.\, \tau$. The type $\texttt{ref}(\tau)$ is the type of memory locations that store values of type τ.

We write $\Xi \mid \Gamma \vdash e : \tau$ to express that expression e has type τ under typing contexts Γ and Ξ. The typing context Γ is a list of the form $x_1 : \tau_1, \cdots, x_n : \tau_n$. It associates free variables (that may appear in e) to their types. The type-level context Ξ is a list, $X_1, X_2, \cdots$, of free type variables (that may appear in τ or Γ). The typing rules of $\mathsf{F}_{\mu,ref,conc}$ are presented in Figure 5.1. Notice that the typing rule for the `CAS` operation has a side-condition EqType which ensures that the `CAS` operation can only be performed on word-sized data types.

We define the small-step operational semantics of $\mathsf{F}_{\mu,ref,conc}$ in two stages. We first define a head reduction relation, $\to_{\mathsf{h}}$. The (thread-pool) reduction relation for our programs, $\to_{\mathsf{tp}}$, is then defined as individual threads taking head steps under some evaluation contexts, K, or threads being forked.

$$\frac{(\sigma, e) \to_{\mathsf{h}} (\sigma', e')}{(\sigma, \vec{e_1}; K[e]; \vec{e_2}) \to_{\mathsf{tp}} (\sigma', \vec{e_1}; K[e']; \vec{e_2})}$$

$$(\sigma, \vec{e_1}; K[\texttt{fork}\ \{e\}]; \vec{e_2}) \to_{\mathsf{tp}} (\sigma, \vec{e_1}; K[()]; \vec{e_2}; e)$$

Here, our state, $\sigma : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$, is the heap which we model as a partial map with finite support from memory locations to values. The evaluation contexts of $\mathsf{F}_{\mu,ref,conc}$ are the following:

$$K ::= [] \mid K \circledcirc e \mid v \circledcirc K \mid K\ e \mid v\ K \mid K\ \_ \mid (K, e) \mid (v, K) \mid \pi_i\, K \mid$$

$$\text{T-VAR}\ \frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau} \qquad \text{T-REC}\ \frac{\Xi \mid \Gamma, x : \tau, f : \tau \to \tau' \vdash e : \tau \to \tau'}{\Xi \mid \Gamma \vdash \texttt{rec}\, f(x) = e : \tau \to \tau'} \qquad \text{T-TLAM}\ \frac{\Xi, X \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall X.\, \tau}$$

$$\text{T-APP}\ \frac{\Xi \mid \Gamma \vdash e_1 : \tau \to \tau' \qquad \Xi \mid \Gamma \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash e_1\ e_2 : \tau'} \qquad \text{T-TAPP}\ \frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]}$$

$$\text{T-INJ}\ \frac{\Xi \mid \Gamma \vdash e : \tau_i \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \texttt{inj}_i\, e : \tau_1 + \tau_2} \qquad \text{T-IF}\ \frac{\Xi \mid \Gamma \vdash e : \mathbb{B} \qquad \Xi \mid \Gamma \vdash e_i : \tau \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau}$$

$$\text{T-PAIR}\ \frac{\Xi \mid \Gamma \vdash e_1 : \tau_1 \qquad \Xi \mid \Gamma \vdash e_2 : \tau_2}{\Xi \mid \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{T-PROJ}\ \frac{\Xi \mid \Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \pi_i\, e : \tau_i}$$

$$\text{T-MATCH}\ \frac{\Xi \mid \Gamma \vdash e : \tau_1 + \tau_2 \qquad \Xi \mid \Gamma, x : \tau_i \vdash e_i : \tau_3 \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \texttt{match}\, e\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} : \tau_3}$$

$$\text{T-FOLD}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau/X]}{\Xi \mid \Gamma \vdash \texttt{fold}\, e : \mu X.\, \tau} \qquad \text{T-UNFOLD}\ \frac{\Xi \mid \Gamma \vdash e : \mu X.\, \tau}{\Xi \mid \Gamma \vdash \texttt{unfold}\, e : \tau[\mu X.\, \tau/X]}$$

$$\text{T-ALLOC}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \texttt{ref}(e) : \texttt{ref}(\tau)} \qquad \text{T-LOAD}\ \frac{\Xi \mid \Gamma \vdash e : \texttt{ref}(\tau)}{\Xi \mid \Gamma \vdash\, !\, e : \tau}$$

$$\text{T-STORE}\ \frac{\Xi \mid \Gamma \vdash e_1 : \texttt{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash e_1 \leftarrow e_2 : 1} \qquad \text{T-FORK}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \texttt{fork}\, \{e\} : 1}$$

$$\text{T-CAS}\ \frac{\Xi \mid \Gamma \vdash e_1 : \texttt{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau \qquad \Xi \mid \Gamma \vdash e_3 : \tau \qquad \mathsf{EqType}(\tau)}{\Xi \mid \Gamma \vdash \texttt{CAS}(e_1, e_2, e_3) : \mathbb{B}}$$

$$\text{EqTyp-unit}\ \mathsf{EqType}(1) \qquad \text{EqTyp-nat}\ \mathsf{EqType}(\mathbb{N}) \qquad \text{EqTyp-bool}\ \mathsf{EqType}(\mathbb{B}) \qquad \text{EqTyp-ref}\ \mathsf{EqType}(\texttt{ref}(\tau))$$

Figure 5.1: The typing rules of $\mathsf{F}_{\mu,ref,conc}$ (some trivial cases omitted)

$$(\sigma, (\texttt{rec}\, f(x) = e)\ v) \to_{\mathsf{h}} (\sigma, e[f, x/\texttt{rec}\, f(x) = e, v]) \qquad (\sigma, (\Lambda\, e)\ \_) \to_{\mathsf{h}} (\sigma, e)$$

$$(\sigma, \texttt{if true then}\, e_1\, \texttt{else}\, e_2) \to_{\mathsf{h}} (\sigma, e_1)$$

$$(\sigma, \texttt{if false then}\, e_1\, \texttt{else}\, e_2) \to_{\mathsf{h}} (\sigma, e_2)$$

$$\frac{i \in \{1, 2\}}{(\sigma, \texttt{match}\, \texttt{inj}_i\, v\, \texttt{with}\, \texttt{inj}_1\, x \Rightarrow e_1 \mid \texttt{inj}_2\, x \Rightarrow e_2\, \texttt{end}) \to_{\mathsf{h}} (\sigma, e_i[v/x])}$$

$$(\sigma, \texttt{unfold}\, (\texttt{fold}\, v)) \to_{\mathsf{h}} (\sigma, v) \qquad \frac{\ell \notin \mathrm{dom}(\sigma)}{(\sigma, \texttt{ref}(v)) \to_{\mathsf{h}} (\sigma \uplus \{(\ell, v)\}, \ell)}$$

$$\frac{(\ell, v) \in \sigma}{(\sigma, !\, \ell) \to_{\mathsf{h}} (\sigma, v)} \qquad \frac{i \in \{1, 2\}}{(\sigma, \pi_i\, (v_1, v_2)) \to_{\mathsf{h}} (\sigma, v_i)}$$

$$(\sigma \uplus \{(\ell, v)\}, \ell \leftarrow w) \to_{\mathsf{h}} (\sigma \uplus \{(\ell, w)\}, ())$$

$$(\sigma \uplus \{(\ell, v)\}, \texttt{CAS}(\ell, v, w)) \to_{\mathsf{h}} (\sigma \uplus \{(\ell, w)\}, \texttt{true})$$

$$\frac{v \neq w}{(\sigma \uplus \{(\ell, v)\}, \texttt{CAS}(\ell, w, u)) \to_{\mathsf{h}} (\sigma \uplus \{(\ell, v)\}, \texttt{false})}$$

Figure 5.2: The rules for the head step relation of $\mathsf{F}_{\mu,ref,conc}$

$$\texttt{if}\, K\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{inj}_i\, K \mid \texttt{match}\, K\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} \mid$$

$$\texttt{fold}\, K \mid \texttt{unfold}\, K \mid \texttt{ref}(K) \mid\ !\, K \mid K \leftarrow e \mid v \leftarrow K \mid$$

$$\texttt{CAS}(K, e, e) \mid \texttt{CAS}(v, K, e) \mid \texttt{CAS}(v, v, K)$$

The evaluation contexts above define a call-by-value (CBV) evaluation strategy for $\mathsf{F}_{\mu,ref,conc}$. The rules for head steps are presented in Figure 5.2. We write $\uplus$ for the disjoint union operation on heaps.

Safety We say a program e is safe, written $\mathit{Safe}(e)$, if it does not get stuck. This is formally defined as follows:

$$\mathit{Safe}(e) \triangleq \forall e', \vec{e_1}, \vec{e_2}, \sigma.\ (\emptyset, e) \to_{\mathsf{tp}}^{*} (\sigma, \vec{e_1}; e'; \vec{e_2}) \Rightarrow$$

$$e' \in \mathit{Val} \vee (\exists e'', \sigma'.\ (\sigma, \vec{e_1}; e'; \vec{e_2}) \to_{\mathsf{tp}} (\sigma', \vec{e_1}; e''; \vec{e_2})) \vee$$

$$(\exists e'', \sigma', e_3.\ (\sigma, \vec{e_1}; e'; \vec{e_2}) \to_{\mathsf{tp}} (\sigma', \vec{e_1}; e''; \vec{e_2}; e_3))$$

In prose, e is safe if any expression e', that it or one of its children threads have reached, e' is either a value or it can be evaluated further by making a head step or by forking a thread.

Contextual refinement and equivalence For a context (a program with a hole) C we write $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')$ to express that C is a well-typed context that takes programs of type τ to programs of type τ'. More precisely, $\Xi' \mid \Gamma' \vdash C[e] : \tau'$ holds for any expression e such that $\Xi \mid \Gamma \vdash e : \tau$.

We formally define contextual refinement and contextual equivalence for programs written in $\mathsf{F}_{\mu,ref,conc}$ as follows:

Definition 5.2.1 (Contextual refinement)**.** *We say* e contextually refines e'*, written* $\Xi \mid \Gamma \vDash e \leq_{ctx} e' : \tau$*, if*

$$\Xi \mid \Gamma \vdash e : \tau \land \Xi \mid \Gamma \vdash e' : \tau \land \forall C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; 1).\ C[e]\downarrow \implies C[e']\downarrow$$

where

$$e\downarrow \triangleq (\emptyset, e) \to^*_{\mathsf{tp}} (\sigma, v; \vec{e})$$

Intuitively, when e contextually refines e' no context can distinguish the situation where e' is replaced by e. Since, if there were such a distinguishing context then it could be extended to a context which would converge when filled with e and would diverge when filled with e'. This, however, violates the contextual refinement. Hence, in a contextual refinement $e \leq_{ctx} e'$ we say the e is the *implementation* side and e' the *specification* side.

Definition 5.2.2 (Contextual equivalence)**.** *We say* e *and* e' *are contextually equivalent,* $\Xi \mid \Gamma \vDash e \approx_{ctx} e' : \tau$*, if*

$$\Xi \mid \Gamma \vDash e \leq_{ctx} e' : \tau \land \Xi \mid \Gamma \vDash e' \leq_{ctx} e : \tau$$

5.3 Iris: a brief introduction

Iris (Jung, Krebbers, Birkedal, and Dreyer, 2016; Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer, 2015; Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017) is a state-of-the-art impredicative higher-order concurrent separation logic. It is a program logic designed for formal reasoning about higher-order concurrent imperative programs. Iris is a logical framework and is thus language agnostic. It can be instantiated to a variety of languages.

Here we present Iris as instantiated with $\mathsf{F}_{\mu,ref,conc}$. For a more leisurely paced read on Iris, see Birkedal and Bizjak (2017). In this section and the rest of this paper we will not present the details of how Iris represents resources. Rather we will focus on how Iris can be used as a program logic to facilitate formalization of logical relations. See Appendix C for details of how resources are represented in Iris.

In Iris one can quantify over the Iris types κ:

$$\kappa ::= 1 \mid \kappa \times \kappa \mid \kappa + \kappa \mid \kappa \to \kappa \mid \mathit{Expr} \mid \mathit{Val} \mid \mathbb{N} \mid \mathbb{B} \mid \kappa \xrightarrow{\text{fin}} \kappa \mid \mathit{iProp} \mid \ldots$$

Here, *Expr* and *Val* are types of expressions and values of $\mathsf{F}_{\mu,ref,conc}$, $\kappa \xrightarrow{\text{fin}} \kappa'$ is the type of partial functions with finite support and finset(κ) is the type of finite subsets of κ. The type *iProp* is the type of Iris propositions. These are the following:

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\ P \mid$$

$$\exists x : \kappa.\ P \mid \triangleright P \mid \mu r.\ P \mid \Box P \mid \boxed{P}^{\mathcal{N}} \mid {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P \mid$$

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\} \mid \ell \mapsto v \mid \{P\}\, e\, \{x.\ P\}_{\mathcal{E}} \mid \ldots$$

Here $(\top, \bot, \wedge, \vee, \Rightarrow, \forall, \exists)$ are the conventional connectives of higher-order logic. The connective $*$ is the separating conjunction. Intuitively, the proposition $P * Q$ holds in case the resources can be *separated* into two *disjoint* parts so that one satisfies P and the other Q. The wand connective, $\mathrel{-\!\!*}$, is to separating conjunction as is $\Rightarrow$ to $\wedge$. That is, $P \mathrel{-\!\!*} Q$ holds for those resources such that when conjoined with a resource satisfying P the result would satisfy Q.

The later modality, $\triangleright$, is used to guard recursively defined predicates. That is, $\mu r.\ P$ is a well-defined guarded-recursive predicate only if all occurrences of r in P appear under a $\triangleright$. For guarded recursive definitions we have that $\mu r.P \dashv\vdash P[\mu r.P/r]$ where $\dashv\vdash$ is the logical equivalence of Iris propositions. The $\triangleright$ modality is an abstraction of step-indexing (Appel and McAllester, 2001; Appel, Melliès, Richards, and Vouillon, 2007; Dreyer, Ahmed, and Birkedal, 2011). In terms of step-indexing $\triangleright P$ holds if P holds a step later; hence the name. In Iris this modality is internally used in definitions of weakest preconditions and to guard impredicative invariants to avoid self-referential paradoxes (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). In our logical relations we use the later modality to define the semantics of the recursive types as guarded-recursive predicates, similarly to (Dreyer, Ahmed, and Birkedal, 2011). For any proposition P, we have that $P \vdash \triangleright P$. The later modality commutes with all of the connectives of higher-order separation logic, including quantifiers.[1] The basic reasoning principle for step-indexes in Iris is

[1] The only caveat is that $\triangleright \exists x : \kappa.\ P \vdash \exists x : \kappa.\ P$ only holds if κ is an inhabited type.

the Löb induction rule:

$$\frac{\text{Löb induction}}{\dfrac{\triangleright P \vdash P}{P}}$$

The principle of Löb induction is very similar to the principle of coinduction. It states that if we can prove P under the assumption that $\triangleright P$ holds, then we can prove P without it.

The *persistence* modality $\Box$ forgets all non-duplicable resources. That is, the proposition $\Box P$, pronounced "persistently P", holds if so does P *and* that P only asserts ownership of duplicable resources. We say a proposition P is *persistent*, written $\mathsf{persistent}(P)$, in case $P \dashv\vdash \Box P$. A persistent proposition P is duplicable: $P \dashv\vdash P * P$. The persistence modality is monotonic, i.e., if $P \vdash Q$ then $\Box P \vdash \Box Q$. We say a predicate Φ is persistent, written $\mathsf{persistent}(\Phi)$, if $\forall x.\, \mathsf{persistent}(\Phi(x))$.

In order to express protocols for verification of concurrent programs Iris features *invariants*. We write $\boxed{P}^{\mathcal{N}}$ and read "invariantly P" for the proposition that states that the proposition P is an invariant. The name $\mathcal{N}$ is the name of the invariant and is used to ensure that invariants are not reused in a nested way which in general is unsound. The proposition $\boxed{P}^{\mathcal{N}}$ merely states the *knowledge* that P holds invariantly and does not assert ownership of resources. Hence invariants are persistent. Invariants can be allocated and opened using fancy update modality explained below.

The update modality, ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}$, is used to express manipulation (allocation, deallocation and update) of resources in Iris.[2] The proposition ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ holds for resources that can be updated so that P would hold. For the sake of simplicity we write $\Rrightarrow_{\mathcal{E}}$ instead of ${}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}$. We write $P \;{}^{\mathcal{E}_1}\!\!\Longrightarrow\!\!\!\!\ast^{\mathcal{E}_2}\; Q$, pronounced P viewshifts to Q, as a shorthand for $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$.

The purpose of masks $\mathcal{E}_1$ and $\mathcal{E}_2$ is for bookkeeping of invariants. We write $\top$ for the mask where no invariant is open. Rules for allocating and opening invariants are as follows:

$$\frac{\text{INV-ALLOC}}{\dfrac{\triangleright P}{\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}}} \qquad \frac{\text{INV-OPEN}}{\dfrac{\boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \in \mathcal{E}}{{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\mathcal{N}\}} (\triangleright P * (\triangleright P \;{}^{\mathcal{E}\setminus\{\mathcal{N}\}}\!\!\Longrightarrow\!\!\!\!\ast^{\mathcal{E}}\; \top))}}$$

The INV-ALLOC states that an invariant can be established by proving it one step later. The rule INV-OPEN states that an invariant can be opened if it has not

[2] This modality is called the fancy update modality in Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017.

already been. When opened, we get the body of the invariant at a later step and also that the invariant can be closed by giving up the body but only at a later step. The later modality is necessary here as otherwise impredicative invariants can lead to inconsistencies as shown by Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017).

We write $\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\}$ for the weakest precondition of expression e with respect to the postcondition P. Here x is a binder for the return value. The weakest precondition of a program e with respect to a postcondition P, written $\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\}$, implies two things: (1) the expression e is safe, that is, it may diverge but it will not get stuck, and (2) whenever e converges to a value v, then $P[v/x]$ holds. In particular, the weakest precondition of a value is simply the postcondition (up to updating of resources):

WP-VALUE
$$\mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\} \dashv\vdash \Rightarrow_{\mathcal{E}} \Phi(v)$$

Whenever x does not appear freely in P we write $\mathsf{wp}_{\mathcal{E}}\, e\, \{P\}$ instead of $\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\}$. When Φ is predicate, we sometimes write $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ instead of $\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ \Phi(x)\}$. The mask $\mathcal{E}$ in $\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\}$ is the set of invariant names that are closed before *and* need to be closed at the end. In practice, the update modality is used internally in the definition of weakest preconditions allowing them to open invariants (for at most the duration of an atomic operation) and update resources during proofs of weakest preconditions of programs. See Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017) for the precise definition of weakest preconditions in Iris. Each thread can rely on all invariants. This can be seen in the rule WP-FORK below:

WP-FORK
$$\frac{\triangleright\, \mathsf{wp}_{\top}\, e\, \{\top\}}{\mathsf{wp}_{\mathcal{E}}\, \texttt{fork}\, \{e\}\, \{x.\ x = ()\}}$$

The rule WP-FORK states that the weakest precondition of $\texttt{fork}\, \{e\}$ with respect to Φ follows from the weakest precondition of the forked thread with respect to the trivial postcondition. Note that the mask for the forked thread allows the forked thread to access all invariants. This rule also shows the thread-locality of the weakest preconditions. That is, in proving weakest preconditions we need only to consider the program one thread at a time.

In addition to thread-local reasoning weakest preconditions facilitate context-local reasoning. This is evident from the rule WP-BIND:

WP-BIND
$$\frac{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ \mathsf{wp}_{\mathcal{E}}\, K[x]\, \{\Phi\}\}}{\mathsf{wp}_{\mathcal{E}}\, K[e]\, \{\Phi\}}$$

This rule allows us to focus on a sub-expression while proving weakest preconditions. This sub-expression is usually the sub-expression that makes a head-step. The following are the weakest precondition rules for basic expressions (except for those pertaining to references given further on) that we derive for $\mathsf{F}_{\mu,ref,conc}$:

WP-REC
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e[f, x/\texttt{rec}\, f(x) = e, v]\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, (\texttt{rec}\, f(x) = e)\; v\, \{\Phi\}}$$

WP-TLAM
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, (\Lambda\, e)\; \_\, \{\Phi\}}$$

WP-IF-TRUE
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e_1\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, \texttt{if true then}\, e_1\, \texttt{else}\, e_2\, \{\Phi\}}$$

WP-IF-FALSE
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, \texttt{if true then}\, e_1\, \texttt{else}\, e_2\, \{\Phi\}}$$

WP-PROJ
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, v_i\, \{\Phi\} \qquad i \in \{1, 2\}}{\mathsf{wp}_{\mathcal{E}}\, \pi_i\, (v_1, v_2)\, \{\Phi\}}$$

WP-MATCH
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e_i[v/x]\, \{\Phi\} \qquad i \in \{1, 2\}}{\mathsf{wp}_{\mathcal{E}}\, \texttt{match inj}_i\, v\, \texttt{with inj}_1\, x \Rightarrow e_1 \mid \texttt{inj}_2\, x \Rightarrow e_2\, \texttt{end}\, \{\Phi\}}$$

WP-FOLD
$$\frac{\triangleright\, \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, \texttt{unfold}\, (\texttt{fold}\, v)\, \{\Phi\}}$$

The *maps-to* proposition, $\ell \mapsto v$, is a proposition defined in Iris based on resources. The intuition is straightforward: $\ell \mapsto v$ means that in the heap the location ℓ is allocated and its value v. Furthermore, it asserts that we *exclusively* own this location:

$$\ell \mapsto v * \ell \mapsto v' \vdash \bot$$

For an explanation of how maps-to propositions are defined see Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017). The weakest precondition rules for manipulation of memory are as follows:

WP-ALLOC
$$\frac{\triangleright(\forall \ell.\; \ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \ell\, \{\Phi\})}{\mathsf{wp}_{\mathcal{E}}\, \texttt{ref}(v)\, \{\Phi\}}$$

WP-LOAD
$$\frac{\triangleright\, \ell \mapsto v \qquad \triangleright(\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\})}{\mathsf{wp}_{\mathcal{E}}\, !\ell\, \{\Phi\}}$$

WP-STORE
$$\frac{\triangleright\, \ell \mapsto v \qquad \triangleright(\ell \mapsto w \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, ()\, \{\Phi\})}{\mathsf{wp}_{\mathcal{E}}\, \ell \leftarrow w\, \{\Phi\}}$$

WP-CAS-SUC
$$\frac{\triangleright\, \ell \mapsto v \qquad \triangleright(\ell \mapsto w \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \texttt{true}\, \{\Phi\})}{\mathsf{wp}_{\mathcal{E}}\, \texttt{CAS}(\ell, v, w)\, \{\Phi\}}$$

$$\text{WP-CAS-FAIL}\quad \frac{\triangleright\, \ell \mapsto v \qquad \triangleright(\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \texttt{false}\, \{\Phi\}) \qquad v \neq w}{\mathsf{wp}_{\mathcal{E}}\, \texttt{CAS}(\ell, w, u)\, \{\Phi\}}$$

Hoare triples in Iris are defined based on weakest preconditions like so:

$$\{P\}\, e\, \{\Phi\}_{\mathcal{E}} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\})$$

Whenever the mask is $\top$, instead of $\mathsf{wp}_{\top}\, e\, \{\Phi\}$ and $\{P\}\, e\, \{\Phi\}_{\top}$ we write $\mathsf{wp}\, e\, \{\Phi\}$ and $\{P\}\, e\, \{\Phi\}$, respectively.

5.4 Unary logical relation for type soundness of $\mathsf{F}_{\mu,ref,conc}$

In this section we define a unary logical relations model for $\mathsf{F}_{\mu,ref,conc}$ and use it to prove the type soundness theorem. That is, for each type we define a logical relation. We show that each well-typed program is in the relation for its type. Furthermore, we show that programs in the logical relation for a type have well-defined behavior, i.e., they do not get stuck. The type soundness theorem is a direct consequence of these two facts.

We define the logical relations in three stages. We first define a relation for *closed* values of a type by induction on types. We then use the value relations to define relations on *closed* expressions. Intuitively, a closed expression is in the relation for a type τ if it is a computation that results in a value that is in the value relation for the type τ. Finally, we define *the logical relation* for (open) expressions based on the expression relations and the value relations above.

In $\mathsf{F}_{\mu,ref,conc}$ features polymorphism. Hence, types can have free type variables. Thus, we index the relations on closed values and expressions with a map, Δ, which assigns a semantic type (a value relation) to each free type variable. That is, for each type τ we define the value relation $[\![\Xi \vdash \tau]\!]_{\Delta} : \mathit{Val} \to \mathit{iProp}$ where $\Delta : \Xi \to \mathit{Val} \to \mathit{iProp}$. The full definition of the value relations for types is given in Figure 5.3. We will discuss them in detail below. Before that we discuss how value relations are extended to expression relation on closed and subsequently open expressions.

Intuitively, a closed expression is in the expression relation for a type τ if it computes a result that is in the value relation for the type τ. We define the expression relation, $[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta} : \mathit{Expr} \to \mathit{iProp}$, using Iris's weakest preconditions, as follows:

$$[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e) \triangleq \mathsf{wp}\, e\, \{[\![\Xi \vdash \tau]\!]_{\Delta}\}$$

In order to formally define the logical relation for open expressions we first define a relation, $[\![\Xi \vdash \cdot]\!]^{\mathcal{G}}_{\Delta}$, for typing contexts. Intuitively, a list a values, $\vec{v}$, is in the relation for a typing context, Γ, if each value in $\vec{v}$ is in the value relation for the type corresponding to it in Γ. The formal definition of the typing-context relation is given below. Here, ϵ is the empty list of values.

$$[\![\Xi \vdash \cdot]\!]^{\mathcal{G}}_{\Delta}(\epsilon) \triangleq \top$$

$$[\![\Xi \vdash \Gamma, x : \tau]\!]^{\mathcal{G}}_{\Delta}(\vec{v}, w) \triangleq [\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v}) * [\![\Xi \vdash \tau]\!]_{\Delta}(w)$$

Let Ξ, Γ, e and τ be such that all free variables of e are in the domain of Γ and all free type variables that appear in Γ or τ are in Ξ. Then, we write $\Xi \mid \Gamma \vDash e : \tau$ to express that the expression e is in the logical relation for type τ under the typing contexts Γ and Ξ. This relation is defined as follows:

$$\Xi \mid \Gamma \vDash e : \tau \triangleq \forall \Delta, \vec{v}.\ [\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v}) \vdash [\![\Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e[\vec{v}/\vec{x}])$$

The value relation for types are given in Figure 5.3. One important aspect of the type system of $\mathsf{F}_{\mu,ref,conc}$ is that it is intuitionistic, i.e., values, e.g., function arguments, can be used multiple times. Thus, it is crucial that the value relation of all types are *persistent* and hence duplicable. The persistence modality and the side-condition $\mathsf{persistent}(\Psi)$ in Figure 5.3 are added to ensure the persistence of value relations.

The value relation for type variables is given by Δ. A value is in the relation for the unit type if it is $()$. A values is in the relation for the type of natural numbers if it is simply a natural number; similarly for booleans. A value is in the relation for the product type if it is a pair of values each in their respective types. A value of the sum type $\tau + \tau'$, on the other hand, is either a value in the relation for τ or one in the relation for τ'. The value relation for recursive types is defined using Iris's guarded recursive predicates. A value is in the relation for a recursive type if it is of the form $\texttt{fold}\, w$ such that the value w is, *one step of the computation later*, in the relation for the recursive type. Notice, however, that unfolding a folded value takes a step of computation. A memory location is in the relation for a reference type, $\texttt{ref}(\tau)$, if it *invariantly* stores a value that is in the value relation for τ.

A value v in the relation for the function type $\tau \to \tau'$ if whenever v is applied to a value w, in the relation for τ, the resulting expression, $v\ w$, is in the *expression* relation for τ'. A value is in the relation for the type $\forall X.\, \tau$ if, when instantiated, the resulting *expression* is in the expression relation for τ where the interpretation for X is taken to be any *persistent* predicate.

Notice that the rather compact definitions above are the precise definitions of logical relations given in full detail and precisely what we have in our

$$[\![\Xi \vdash X]\!]_\Delta \triangleq \Delta(X)$$

$$[\![\Xi \vdash 1]\!]_\Delta(v) \triangleq v = ()$$

$$[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v) \triangleq \exists n \in \mathbb{N}.\, v = n$$

$$[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v) \triangleq v \in \{\texttt{true}, \texttt{false}\}$$

$$[\![\Xi \vdash \tau_1 \times \tau_2]\!]_\Delta(v) \triangleq \exists v_1, v_2.\, v = (v_1, v_2) * [\![\Xi \vdash \tau_1]\!]_\Delta(v_1) * [\![\Xi \vdash \tau_2]\!]_\Delta(v_2)$$

$$[\![\Xi \vdash \tau_1 + \tau_2]\!]_\Delta(v) \triangleq \bigvee_{i \in \{1,2\}} \exists w.\, v = \texttt{inj}_i\, w * [\![\Xi \vdash \tau_i]\!]_\Delta(w)$$

$$[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v) \triangleq \Box\left(\forall w.\, [\![\Xi \vdash \tau]\!]_\Delta(w) \mathrel{-\!\!*} [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta(v\ w)\right)$$

$$[\![\Xi \vdash \mu X.\, \tau]\!]_\Delta(v) \triangleq \mu\Psi.\, \exists w.\, v = \texttt{fold}\, w \wedge \triangleright [\![\Xi \vdash \tau]\!]_{\Delta, X \mapsto \Psi}(w)$$

$$[\![\Xi \vdash \forall X.\, \tau]\!]_\Delta(v) \triangleq \Box\left(\forall \Psi.\, \mathsf{persistent}(\Psi) \Rightarrow [\![\Xi, X \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X \mapsto \Psi}(v\ \_)\right)$$

$$[\![\Xi \vdash \texttt{ref}(\tau)]\!]_\Delta(v) \triangleq \exists \ell.\, v = \ell \wedge \boxed{\exists w.\, \ell \mapsto w * [\![\Xi \vdash \tau]\!]_\Delta(w)}^{\mathcal{N}.\ell}$$

Figure 5.3: The unary value relation for types of $\mathsf{F}_{\mu,ref,conc}$

mechanization in Coq. These definitions are very similar in size and style to what one would write for, say, simply typed lambda calculus or System F.

Lemma 5.4.1. *Let τ be a type such that all its free type variables appear in Ξ. Furthermore, let X be a type variable such that $X \notin \Xi$ and let Δ be an interpretation for type variables in Ξ. It follows that*

$$[\![\Xi \vdash \tau]\!]_\Delta(v) \dashv\vdash [\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto \Psi}(v)$$

for any predicate Ψ and value v.

Proof. By induction on the structure of τ. □

Lemma 5.4.2. *Let Γ be a typing context such that all free type variables of Γ appear in Ξ. Furthermore, let X be a type variable such that $X \notin \Xi$ and let Δ be an interpretation for type variables in Ξ. It follows that*

$$[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v}) \dashv\vdash [\![\Xi, X \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta, X \mapsto \Psi}(\vec{v})$$

for any predicate Ψ and sequence of values $\vec{v}$.

Proof. By induction on the length of Γ using Lemma 5.4.1. □

Theorem 5.4.3 (Fundamental theorem of unary logical relations)**.** *All well-typed terms are in the logical relation.*

$$\textit{If}\ \ \Xi \mid \Gamma \vdash e : \tau \ \ \textit{then}\ \ \Xi \mid \Gamma \vDash e : \tau$$

Proof. By induction on the typing derivation. All cases follow from the inference rules of weakest preconditions presented in Section 5.3. Here, we present a few cases of this proof.

– Case T-ALLOC: For this case, given $\Delta : \Xi \to \mathit{Val} \to \mathit{iProp}$ and a list of values $\vec{v}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v})$, we need to show assuming

$$\mathsf{wp}\, e[\vec{v}/\vec{x}]\, \{[\![\Xi \vdash \tau]\!]_{\Delta}\} \tag{5.1}$$

that the following holds:

$$\mathsf{wp}\, \texttt{ref}(e[\vec{v}/\vec{x}])\, \left\{x.\exists \ell.\ x = \ell \wedge \boxed{\exists w.\ \ell \mapsto w * [\![\Xi \vdash \tau]\!]_{\Delta}(w)}^{\mathcal{N}.\ell}\right\}$$

We use the rule WP-BIND together with the assumption (5.1) above. Consequently, we need to show that given some arbitrary value v such that

$$[\![\Xi \vdash \tau]\!]_{\Delta}(v) \tag{5.2}$$

we have

$$\mathsf{wp}\, \texttt{ref}(v)\, \left\{x.\exists \ell.\ x = \ell \wedge \boxed{\exists w.\ \ell \mapsto w * [\![\Xi \vdash \tau]\!]_{\Delta}(w)}^{\mathcal{N}.\ell}\right\}$$

We proceed by applying the rule WP-ALLOC which requires us to show:

$$\triangleright \forall \ell'.\ \ell' \mapsto v \mathrel{-\!\!*} \mathsf{wp}\, \ell'\, \left\{x.\exists \ell.\ x = \ell \wedge \boxed{\exists w.\ \ell \mapsto w * [\![\Xi \vdash \tau]\!]_{\Delta}(w)}^{\mathcal{N}.\ell}\right\}$$

This follows easily from the rules WP-VALUE and INV-ALLOC together with assumption (5.2) above.

– Case T-REC: For this case, given $\Delta : \Xi \to \mathit{Val} \to \mathit{iProp}$ and a list of values $\vec{v}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v})$, we need to show assuming

$$\forall v, u.\ [\![\Xi \vdash \tau]\!]_{\Delta}(v) * [\![\Xi \vdash \tau \to \tau']\!]_{\Delta}(u) \mathrel{-\!\!*}$$
$$\mathsf{wp}\, (e[\vec{v}/\vec{x}])[v, u/x, f]\, \{[\![\Xi \vdash \tau']\!]_{\Delta}\} \tag{5.3}$$

that the following holds:

$$\mathsf{wp}\,\texttt{rec}\,f(x) = e[\vec{v}/\vec{x}]\,\{x.\ \Box\,(\forall w.\ [\![\Xi \vdash \tau]\!]_\Delta(w) \twoheadrightarrow [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta(x\ w))\}$$

By the rule WP-VALUE it suffices to show:[3]

$$\forall w.\ [\![\Xi \vdash \tau]\!]_\Delta(w) \twoheadrightarrow [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta((\texttt{rec}\,f(x) = e[\vec{v}/\vec{x}])\ w)$$

Note the operational semantics pertaining to calling a recursive functions. The expression

$$(\texttt{rec}\,f(x) = e[\vec{v}/\vec{x}])\ w$$

reduces to the expression

$$(e[\vec{v}/\vec{x}])[w, \texttt{rec}\,f(x) = e[\vec{v}/\vec{x}]/x, f]$$

in a single step of computation. Hence, if we know that $[\![\Xi \vdash \tau \to \tau']\!]_\Delta(\texttt{rec}\,f(x) = e[\vec{v}/\vec{x}])$ holds we can use the assumption (5.3) above to finish the proof. However, this is exactly what we have to show but crucially we only need this *after one step of computation*, intuitively because a recursive call can only occur inside the body, after the current call. Hence, to finish the proof we use the LÖB INDUCTION rule. Consequently we get to assume the following Löb induction hypothesis (IH).

$$\triangleright \forall w.\ [\![\Xi \vdash \tau]\!]_\Delta(w) \twoheadrightarrow [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta((\texttt{rec}\,f(x) = e[\vec{v}/\vec{x}])\ w) \qquad \text{(IH)}$$

We finish the proof using the rule WP-REC which requires us to show the following for some arbitrary value w for which we have $[\![\Xi \vdash \tau]\!]_\Delta(w)$:

$$\triangleright\, \mathsf{wp}\,(e[\vec{v}/\vec{x}])[w, \texttt{rec}\,f(x) = e[\vec{v}/\vec{x}]/x, f]\,\{[\![\Xi \vdash \tau']\!]_\Delta\}$$

This follows easily from our assumptions and the Löb induction hypothesis, (IH), above.

– Case T-TLAM: For this case, given $\Delta : \Xi \to \mathit{Val} \to \mathit{iProp}$ and a list of values $\vec{v}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v})$, we need to show assuming

$$\forall \Psi.\ [\![\Xi, X \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta, X \mapsto \Psi}(\vec{v}) \vdash \mathsf{wp}\,e[\vec{v}/\vec{x}]\,\{[\![\Xi, X \vdash \tau']\!]_{\Delta, X \mapsto \Psi}\} \qquad (5.4)$$

that the following holds:

$$\mathsf{wp}\,\Lambda\,e[\vec{v}/\vec{x}]\,\{x.\ \Box\,(\forall \Psi.\ \mathsf{persistent}(\Psi) \Rightarrow [\![\Xi, X \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X \mapsto \Psi}(v\ \_))\}$$

[3] We can introduce the persistence modality because all assumptions are persistent.

Since $\Lambda\, e[\vec{v}/\vec{x}]$ is a value, we use the rule WP-VALUE. Hence, it suffices to show the following for some arbitrary but fixed Ψ such that $\mathsf{persistent}(\Psi)$:

$$[\![\Xi, X \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X \mapsto \Psi}((\Lambda\, e[\vec{v}/\vec{x}])\ \_)$$

Note that here we can introduce the persistence modality as none of our assumptions assert any ownership. Unfolding the expression relation in the above formula reveals that we need to show:

$$\mathsf{wp}\ (\Lambda\, e[\vec{v}/\vec{x}])\ \_\{[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto \Psi}\}$$

To prove this, we proceed by applying the rule WP-TLAM and as a result need to show:

$$\triangleright\, \mathsf{wp}\ e[\vec{v}/\vec{x}]\,\{[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto \Psi}\}$$

Finally, we can finish the proof by appealing to assumption (5.4). We only need to show $[\![\Xi, X \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta, X \mapsto \Psi}(\vec{v})$ while we have $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v})$. However, this follows from Lemma 5.4.2.

□

Lemma 5.4.4 (Adequacy of unary logical relations)**.** *Let e be an expression such that $[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e)$. Then, e is safe, Safe(e).*

Proof. This lemma is a direct consequence of the adequacy theorem for Iris's weakest preconditions (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). □

Theorem 5.4.5 (Soundness of unary logical relations)**.** *All closed well-typed programs of $\mathsf{F}_{\mu,ref,conc}$ are safe:*

$$\textit{If}\quad \cdot \mid \cdot \vdash e : \tau \quad \textit{then}\quad \textit{Safe}(e)$$

Proof. By the fundamental theorem of unary logical relation, Theorem 5.4.3, we know that

$$\cdot \mid \cdot \vDash e : \tau$$

Expanding the definition of unary logical relations we get

$$\forall \Delta, \vec{v}.\ [\![\cdot \vdash \cdot]\!]^{\mathcal{G}}_{\Delta}(\vec{v}) \vdash [\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\Delta}\ (e[\vec{v}/\vec{x}])$$

We take $\Delta = \emptyset$ and $\vec{v} = \epsilon$ which gives us

$$[\![\cdot \vdash \cdot]\!]^{\mathcal{G}}_{\emptyset}(\epsilon) \vdash [\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\Delta}\ (e[\epsilon/\epsilon])$$

which immediately simplifies to $[\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e)$. By Theorem 5.4.4, we get *Safe*(e), as required. □

The logical relations model presented in this section is modular. For instance, a well-typed function of type $\tau \to \tau'$ (which by the fundamental theorem falls in the logical relation) can be applied to any expression that is the logical relations for τ and the result is in the logical relation for τ'. On the other hand, our logical relations model is defined in terms of untyped expressions. This means that our logical relations model can be used to prove safety of programs that mix well-typed code with untyped code as long as we show that the untyped code is *semantically well-typed*, i.e., the untyped code is in the logical relations for the appropriate type. As an example of a program that is *not* well-typed but is nonetheless semantically well-typed consider the following:

$$\Lambda\,\Lambda\,(\lambda f.\,\lambda x.\,\lambda g.\,\lambda y.\,\mathtt{let}\,l = \mathtt{ref}(\mathtt{true})\,\mathtt{in}\,\mathtt{fork}\,\{l \leftarrow \mathtt{false}\};$$
$$\mathtt{if}\,!l\,\mathtt{then}\,\mathit{waitfor}\;l; l \leftarrow x; \mathtt{inj}_1\,(f\;l)\,\mathtt{else}\,l \leftarrow y; \mathtt{inj}_2\,(g\;l))$$

where

$$\mathit{waitfor} \triangleq \mathtt{rec}\,f(x) = \mathtt{if}\,!x\,\mathtt{then}\,f\;x\,\mathtt{else}\,()$$

This program does not syntactically have the type

$$\forall X.\,\forall Y.\,(\mathtt{ref}(X) \to X) \to X \to (\mathtt{ref}(Y) \to Y) \to Y \to X + Y$$

but it *semantically* does. This program allocates a boolean reference and uses it to non-deterministically call f or g. In each case it changes the reference that it has already allocated with the given value of the appropriate type before passing it to the chosen function. In case the decision is made to call the first function, i.e., the other thread has not succeeded in the race, it waits for the other thread to finish. This is to ensure that the other thread writing to the reference l is not going to destroy the contents of l at some later point. Despite not being syntactically well-typed one can show that the program above is semantically of the type given. Hence, this program can be *safely* linked against any other (syntactically or semantically) well-typed program with compatible type. See Appendix D.1 for a sketch of a proof that this program is indeed semantically well-typed.

5.5 Binary logical relation for contextual refinements

In this section we define a binary logical relations model for $\mathsf{F}_{\mu,ref,conc}$ and show how to use that for proving contextual refinement of programs. That is, we define a binary logical relations model such that being in the logical

relations implies contextual refinement. For this reason we also refer to our binary logical relation as logical refinement. The main challenge that we address in this section is how to use Iris, a program logic with no support for relational reasoning about programs, to construct a *binary* logical relations model. We solve this problem by introducing propositions in Iris, defined based on Iris resources, that allow us to reason about the execution on the specification side of a logical refinement. We use Iris's weakest preconditions to reason about the implementation side of a logical refinement.

The key idea in reasoning about the specification side is to be able to refer to (the heap and different threads of) the program on the specification side "that is about to be executed". The intuitive definition of two expressions being related is as follows:

> *An expression e (the implementation side) is related to an expression e' (the specification side) if we have:*
>
> $$\forall j, K.\ \text{“thread } j \text{ is about to execute } K[e']\text{”} \ \twoheadrightarrow$$
>
> $$\mathsf{wp}\, e\, \{\exists v'.\ \text{“thread } j \text{ is about to execute } K[v']\text{”}\}$$

This relation between e and e' reads as follows: if thread j is about to execute e' under some evaluation context K on the specification side and e reduces to a value, then, there is a value v' such that the specification side is about to execute $K[v']$. In other words, whenever e reduces to a value we know that e' has also been reduced to v'. The reason for explicit quantification over the thread j under which the specification side is being executed is to enable thread-local reasoning. We quantify over the evaluation context K under which the expression is about to be executed to enable modular reasoning with respect to evaluation contexts.

To reason about the execution on the specification side we use Iris resources and invariants. Here we only discuss the resources that are needed for our binary logical relation at a high level of abstraction, and just specify the axioms and rules that we require these resources to satisfy. For details of how these resources are constructed in Iris out of basic building blocks of resources see Appendix D.2.

We use the following propositions in Iris to track the execution on the specification side.

$$j \Rrightarrow e \qquad \ell \mapsto_s v \qquad \mathrm{SpecConf}(\rho)$$

Here, ρ is a configuration, i.e., a pair of a heap and a thread pool, $(\sigma, \vec{e})$. The intuition is that $\mathrm{SpecConf}(\rho)$ is the configuration that the specification side is

in. These propositions are, crucially, exclusive:

$$\mathrm{SpecConf}(\rho) * \mathrm{SpecConf}(\rho') \vdash \bot \qquad \ell \mapsto_s v * \ell \mapsto_s v' \vdash \bot$$

$$j \Mapsto e * j \Mapsto e' \vdash \bot$$

That is, the configuration, which the specification side is in, is uniquely determined by the proposition $\mathrm{SpecConf}(\rho)$. The propositions $j \Mapsto e$ and $\ell \mapsto_s v$ simply specify the exclusive ownership of the execution of a thread or a memory location on the specification side, allowing for modular reasoning about the specification side. The following rules hold for these resources:

SPEC-THREAD
$$\frac{\mathrm{SpecConf}(\sigma, \vec{e_1}; e; \vec{e_2}) \qquad j \Mapsto e' \qquad j = \mathsf{len}(\vec{e_1})}{e = e'}$$

SPEC-HEAP
$$\frac{\mathrm{SpecConf}(\sigma, \vec{e}) \qquad \ell \mapsto_s v}{\sigma(\ell) = v}$$

SPEC-THREAD-UPD
$$\frac{\mathrm{SpecConf}(\sigma, \vec{e_1}; e; \vec{e_2}) \qquad j \Mapsto e \qquad j = \mathsf{len}(\vec{e_1})}{\Rrightarrow_{\mathcal{E}} \mathrm{SpecConf}(\sigma, \vec{e_1}; e'; \vec{e_2}) * j \Mapsto e'}$$

SPEC-HEAP-UPD
$$\frac{\mathrm{SpecConf}(\sigma \uplus \{(\ell, v)\}, \vec{e}) \qquad \ell \mapsto_s v}{\Rrightarrow_{\mathcal{E}} \mathrm{SpecConf}(\sigma \uplus \{(\ell, v')\}, \vec{e}) * \ell \mapsto_s v'}$$

SPEC-THREAD-ALLOC
$$\frac{\mathrm{SpecConf}(\sigma, \vec{e_1})}{\Rrightarrow_{\mathcal{E}} \mathrm{SpecConf}(\sigma, \vec{e_1}; e) * \mathsf{len}(\vec{e_1}) \Mapsto e}$$

SPEC-HEAP-ALLOC
$$\frac{\mathrm{SpecConf}(\sigma, \vec{e}) \qquad \ell \notin \mathrm{dom}(\sigma)}{\Rrightarrow_{\mathcal{E}} \mathrm{SpecConf}(\sigma \uplus \{(\ell, v)\}, \vec{e}) * \ell \mapsto_s v}$$

The rules above capture the intuitive nature of the predicates $\mathrm{SpecConf}(\rho)$, $j \Mapsto e$ and $\ell \mapsto_s v$. That is, $\mathrm{SpecConf}(\rho)$ tracks evaluation on the specification side, while $j \Mapsto e$ and $\ell \mapsto_s v$ express exclusive ownership of the a thread and a memory location, respectively.

As we explained above, we need to reason about the state of the execution on the specification side. For this purpose we will use an invariant which in turn uses the $\mathrm{SpecConf}(\rho)$ proposition to express that the configuration ρ is reachable from *the starting configuration*. As we reason modularly, we do not know the starting point of the execution for the specification side. In particular, when proving contextual refinements, the starting point of the execution for the specification depends on the particular well-typed context being considered. Therefore, in order to express that the specification side is in a configuration reachable from some starting configuration ρ, we use the invariant $\mathrm{SpecCtx}(\rho)$:

$$\mathrm{SpecCtx}(\rho) \triangleq \boxed{\exists \rho'.\ \mathrm{SpecConf}(\rho') \land \rho \to_{\mathsf{tp}}^* \rho'}^{\mathcal{N}.sc}$$

The following rules can be derived easily for execution on the specification side using the rules for invariants together with the rules SPEC-THREAD, SPEC-HEAP, SPEC-THREAD-UPD, SPEC-HEAP-UPD, SPEC-THREAD-ALLOC and SPEC-HEAP-ALLOC.

SPEC-REC
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[(\texttt{rec}\, f(x) = e)\ v]}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[e[f, x/\texttt{rec}\, f(x) = e, v]]}$$

SPEC-TLAM
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto [(\Lambda\, e)\ \_]}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[e]}$$

SPEC-IF-TRUE
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[\texttt{if true then}\, e_1\, \texttt{else}\, e_2]}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[e_1]}$$

SPEC-IF-FALSE
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[\texttt{if true then}\, e_1\, \texttt{else}\, e_2]}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[e_2]}$$

SPEC-PROJ
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[\pi_i\, (v_1, v_2)] \qquad i \in \{1, 2\}}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[v_i]}$$

SPEC-MATCH
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[\texttt{match inj}_i\, v\, \texttt{with inj}_1\, x \Rightarrow e_1 \mid \texttt{inj}_2\, x \Rightarrow e_2\, \texttt{end}] \qquad i \in \{1, 2\}}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[e_i]}$$

SPEC-FOLD
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[\texttt{unfold}\, (\texttt{fold}\, v)]}{\models\!\!\Rightarrow_{\mathcal{E}} j \Mapsto K[v]}$$

SPEC-ALLOC
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[\texttt{ref}(v)]}{\models\!\!\Rightarrow_{\mathcal{E}} \exists \ell.\ \ell \mapsto_s v * j \Mapsto K[\ell]}$$

SPEC-LOAD
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Mapsto K[!\,\ell] \qquad \ell \mapsto_s v}{\models\!\!\Rightarrow_{\mathcal{E}} \ell \mapsto_s v * j \Mapsto K[v]}$$

SPEC-STORE
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad \ell \mapsto_s v \qquad j \Mapsto K[\ell \leftarrow w]}{\models\!\!\Rightarrow_{\mathcal{E}} \ell \mapsto_s w * j \Mapsto K[()]}$$

SPEC-CAS-SUC
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad \ell \mapsto_s v \qquad j \Rightarrow K[\mathtt{CAS}(\ell, v, w)]}{\Rrightarrow_{\mathcal{E}} \ell \mapsto_s w * j \Rightarrow K[\mathtt{true}]}$$

SPEC-CAS-FAIL
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad \ell \mapsto_s v \qquad j \Rightarrow K[\mathtt{CAS}(\ell, w, u)] \qquad v \neq w}{\Rrightarrow_{\mathcal{E}} \ell \mapsto_a v * j \Rightarrow K[\mathtt{false}]}$$

SPEC-FORK
$$\frac{\mathcal{N}.sc \in \mathcal{E} \qquad \mathrm{SpecCtx}(\rho) \qquad j \Rightarrow K[\mathtt{fork}\ \{e\}]}{\Rrightarrow_{\mathcal{E}} \exists j'.\ j \Rightarrow K[()] * j' \Rightarrow e}$$

The binary logical relations

Similar to the unary logical relations above for $\mathsf{F}_{\mu,ref,conc}$, we define the binary logical relations for $\mathsf{F}_{\mu,ref,conc}$ in three stages. We first define (by induction on τ) a binary relation $[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ on closed values. Here, $\Delta : \Xi \to \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ is the binary value relation for the free type variables in Ξ. Next, we use the binary value relation of a type τ to define the binary relation for τ on closed expressions, written $[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$. Finally, we use the value and expression relations above on closed terms to define the binary logical relations on open expressions.

The formal definition of the expression relation is similar to the intuitive definition that we gave at the beginning of this section. We formalize the phrase "thread j is about to execute ..." using the propositions that we introduced for reasoning about the execution on the specification side. The binary expression relation, $[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta$, is defined as follows:

$$[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e, e') \triangleq \forall \rho, j, K.\ \mathrm{SpecCtx}(\rho) * j \Rightarrow K[e'] \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\ \exists v'.\ j \Rightarrow K[v'] * [\![\Xi \vdash \tau]\!]_\Delta(v, v')\}$$

The expression relation above states that e and e' are related if the following holds: for any thread j which is about to execute e' under some evaluation context K, it is safe to evaluate e and whenever e reduces to a value v, we know that e' has also been evaluated to some value v' in thread j under the evaluation context K. Furthermore, we know that v and v' will be related as values of the type relating e and e'. Thus, essentially, two expressions e and e' are related at type τ if, whenever e reduces to a value, so does e' (no matter

under which circumstances it is being evaluated), and the resulting values will be related at type τ.

Similar to what we did for the unary logical relations, in order to define the logical relation on open expressions, we define relations for typing contexts:

$$[\![\Xi \vdash \cdot]\!]^{\mathcal{G}}_{\Delta}(\epsilon) \triangleq \top$$

$$[\![\Xi \vdash \Gamma, x : \tau]\!]^{\mathcal{G}}_{\Delta}((\vec{v}, w), (\vec{v'}, w')) \triangleq [\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v}, \vec{v'}) * [\![\Xi \vdash \tau]\!]_{\Delta}(w, w')$$

We define the binary logical relations for $\mathsf{F}_{\mu,ref,conc}$, written $\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau$, as follows:

$$\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau \triangleq \forall \vec{v}, \vec{v'}, \Delta.\ [\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v}, \vec{v'}) \vdash [\![\Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$

where $\vec{x}$ is the domain of Γ.

The binary value relation for types are defined in Figure 5.4. The binary value relations, similarly to unary value relations, need to be persistent. This is due to the fact that the intuitionistic type system of $\mathsf{F}_{\mu,ref,conc}$ allows for values to be used multiple times. This is the reason for the persistence modality and the side condition $\mathsf{persistent}(\Psi)$ in Figure 5.4.

The binary value relations in Figure 5.4 are straightforward extensions of their unary counter parts. Values of a base types (1, $\mathbb{N}$ and $\mathbb{B}$), are related if they are equal. A pair of values are related at the product type, if both are themselves pairs of values, which are component-wise in the value interpretation of the corresponding type. The interpretation of sum types relates a pair of values if they are both constructed using the same injection with underlying values related at the corresponding type. A pair of values are related as functions if applying them to values related at the domain type produces expressions related at the codomain type. A pair of values v and v' are in the relation for a recursive type if v is of the form $\texttt{fold}\, w$ and v' is of the form $\texttt{fold}\, w'$, such that the values w and w' are, *one step of the computation later*, in the relation for the recursive type. Notice however that unfolding a folded value takes a step of computation. A pair of values are related at a polymorphic type if we have that the specialization of the two polymorphic types are related, regardless of which (persistent) predicate we take as the value interpretation of the quantified type. Finally, two locations are related at a reference type, $\texttt{ref}(\tau)$, if they invariantly store values that are related at τ.

Lemma 5.5.1 (Congruence of binary logical relations w.r.t. typing)**.** *Our logical relations is a congruence relation with respect to all typing rules. Being a congruence relation w.r.t. the rules* T-ALLOC*,* T-REC *and* T-TLAM *amounts to the following:*

$$\llbracket \Xi \vdash X \rrbracket_\Delta \triangleq \Delta(X)$$

$$\llbracket \Xi \vdash 1 \rrbracket_\Delta(v, v') \triangleq v = v' = ()$$

$$\llbracket \Xi \vdash \mathbb{N} \rrbracket_\Delta(v, v') \triangleq \exists n \in \mathbb{N}.\ v = v' = n$$

$$\llbracket \Xi \vdash \mathbb{B} \rrbracket_\Delta(v, v') \triangleq v = v' \in \{\texttt{true}, \texttt{false}\}$$

$$\llbracket \Xi \vdash \tau_1 \times \tau_2 \rrbracket_\Delta(v) \triangleq \exists v_1, v_2, v_1', v_2'.\ v = (v_1, v_2) * v' = (v_1', v_2') * \llbracket \Xi \vdash \tau_1 \rrbracket_\Delta(v_1, v_1') * \llbracket \Xi \vdash \tau_2 \rrbracket_\Delta(v_2, v_2')$$

$$\llbracket \Xi \vdash \tau_1 + \tau_2 \rrbracket_\Delta(v, v') \triangleq \bigvee_{i \in \{1,2\}} \exists w, w'.\ v = \texttt{inj}_i\, w * v' = \texttt{inj}_i\, w' * \llbracket \Xi \vdash \tau_i \rrbracket_\Delta(w, w')$$

$$\llbracket \Xi \vdash \tau \to \tau' \rrbracket_\Delta(v, v') \triangleq \square\left(\forall w, w'.\ \llbracket \Xi \vdash \tau \rrbracket_\Delta(w, w') \mathrel{-\!\!*} \llbracket \Xi \vdash \tau' \rrbracket^{\mathcal{E}}_\Delta(v\ w, v\ w')\right)$$

$$\llbracket \Xi \vdash \mu X.\, \tau \rrbracket_\Delta(v, v') \triangleq \mu\Psi.\ \exists w, w'.\ v = \texttt{fold}\, w \wedge v' = \texttt{fold}\, w' \wedge \triangleright \llbracket \Xi \vdash \tau \rrbracket_{\Delta, X \mapsto \Psi}(w, w')$$

$$\llbracket \Xi \vdash \forall X.\, \tau \rrbracket_\Delta(v, v') \triangleq \square\left(\forall \Psi.\ \mathsf{persistent}(\Psi) \Rightarrow \llbracket \Xi, X \vdash \tau \rrbracket^{\mathcal{E}}_{\Delta, X \mapsto \Psi}(v\ \_, v'\ \_)\right)$$

$$\llbracket \Xi \vdash \texttt{ref}(\tau) \rrbracket_\Delta(v, v') \triangleq \exists \ell, \ell'.\ v = \ell \wedge v' = \ell' \wedge \boxed{\exists w, w'.\ \ell \mapsto w * \ell' \mapsto_s w' * \llbracket \Xi \vdash \tau \rrbracket_\Delta(w, w')}^{\mathcal{N}.\ell.\ell'}$$

Figure 5.4: The binary value relation for types of $\mathsf{F}_{\mu,ref,conc}$

$$\frac{\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau}{\Xi \mid \Gamma \vDash \texttt{ref}(e) \leq_{log} \texttt{ref}(e') : \texttt{ref}(\tau)}$$

$$\frac{\Xi \mid \Gamma, x : \tau, f : \tau \to \tau' \vDash e \leq_{log} e' : \tau \to \tau'}{\Xi \mid \Gamma \vDash \texttt{rec}\, f(x) = e \leq_{log} \texttt{rec}\, f(x) = e' : \tau \to \tau'}$$

$$\frac{\Xi, X \mid \Gamma \vDash e \leq_{log} e' : \tau}{\Xi \mid \Gamma \vDash \Lambda\, e \leq_{log} \Lambda\, e' : \forall X.\, \tau}$$

Proof. We prove each case separately using the rules for weakest preconditions and evaluation on the specification side. Here we present a few cases of this proof. These are the cases that we proved in the proof of the fundamental theorem of unary logical relations, Theorem 5.4.3. The high-level argument for each case is similar to the corresponding case in the proof of Theorem 5.4.3 with the difference that here we have take into account the execution on the specification side.

– Case T-ALLOC: For this case, given $\Delta : \Xi \to \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ and two lists of values $\vec{v}$ and $\vec{v'}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v}, \vec{v'})$, we need to show assuming

$$\mathrm{SpecCtx}(\rho) \tag{5.5}$$

$$j \Rrightarrow K[\texttt{ref}(e'[\vec{v'}/\vec{x}])] \tag{5.6}$$

$$\forall K'.\, j \Rrightarrow K'[e'[\vec{v'}/\vec{x}]] \mathrel{-\!\!*} \mathsf{wp}\, e[\vec{v}/\vec{x}]\, \{w.\, \exists w'.\, j \Rrightarrow K'[w'] * [\![\Xi \vdash \tau]\!]_{\Delta}(w, w')\} \tag{5.7}$$

that the following holds:

$$\mathsf{wp}\, \texttt{ref}(e[\vec{v}/\vec{x}])\, \{w.\, \exists w'.\, j \Rrightarrow K[w'] * [\![\Xi \vdash \texttt{ref}(\tau)]\!]_{\Delta}(w, w')\}$$

The assumption (5.5) above is persistent. Furthermore, it is clear exactly at which steps of the proof this assumption is used. Hence, during the proofs below we never explicitly mention its uses. We proceed by using the rule WP-BIND, (5.6) and (5.7) (with K' being $K[\texttt{ref}([])]$). As a result we know:

$$[\![\Xi \vdash \tau]\!]_{\Delta}(v, v') \tag{5.8}$$

$$j \Rrightarrow K[\texttt{ref}(v')] \tag{5.9}$$

and have to show:

$$\mathsf{wp}\,\mathtt{ref}(v)\,\{w.\ \exists w'.\ j \Rrightarrow K[w'] * [\![\Xi \vdash \mathtt{ref}(\tau)]\!]_\Delta(w, w')\}$$

Now we use the rule SPEC-ALLOC together with (5.9) to get

$$j \Rrightarrow K[\ell'] \tag{5.10}$$

$$\ell' \mapsto_s v' \tag{5.11}$$

Subsequently, we use the rule WP-ALLOC which gives us

$$\triangleright\, \ell \mapsto v \tag{5.12}$$

and leaves us to prove

$$\triangleright\, \mathsf{wp}\,\ell\,\{w.\ \exists w'.\ j \Rrightarrow K[w'] * [\![\Xi \vdash \mathtt{ref}(\tau)]\!]_\Delta(w, w')\}$$

This now follows trivially after we allocate the invariant $\mathcal{N}.\ell.\ell'$ required to establish $[\![\Xi \vdash \mathtt{ref}(\tau)]\!]_\Delta(\ell, \ell')$.

– Case T-REC: The high-level argument for this case is similar to the proof of the corresponding case in Theorem 5.4.3. The only additional work required is to take into account the execution on the specification side. We only note that the Löb induction hypothesis that is needed to prove this case (see proof of the case T-REC in Theorem 5.4.3) is as follows:

$$\triangleright \forall w, w'.\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \mathrel{-\!\!*}$$

$$[\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta((\mathtt{rec}\, f(x) = e[\vec{v}/\vec{x}])\ w, (\mathtt{rec}\, f(x) = e'[\vec{v'}/\vec{x}])\ w')$$

– Case T-TLAM: The high-level argument for this case is similar to the proof of the corresponding case in Theorem 5.4.3. We only note that binary analogues of Lemmas 5.4.1 and 5.4.2 are needed for the proof of this case.

□

Theorem 5.5.2 (Fundamental theorem of binary logical relations)**.** *All well-typed terms are in the logical relation.*

$$\textit{If}\ \ \Xi \mid \Gamma \vdash e : \tau\ \ \textit{then}\ \ \Xi \mid \Gamma \vDash e \leq_{log} e : \tau$$

Proof. By induction on the typing derivation. Each case, follows from the corresponding congruence lemma in Lemma 5.5.1. □

Lemma 5.5.3 (Congruency of binary logical relations)**.** *Let e and e' be two expressions such that $\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau$ and let $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')$ be a well-typed context. Then, we have:*

$$\Xi' \mid \Gamma' \vDash C[e] \leq_{log} C[e'] : \tau'$$

Proof. By induction on the derivation of $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')$, in each case applying the appropriate compatibility lemma and using the fundamental theorem, Theorem 5.5.2, to show that the well-typed expressions in the context C are related to themselves. □

Lemma 5.5.4 (Adequacy of binary logical relations)**.** *Let e and e' be two expressions such that $[\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\epsilon}(e, e')$ and assume that we have $e \downarrow$. It follows that $e' \downarrow$.*

Proof. We have, by expanding the definition of the expression relation, that

$$\forall \rho, j, K.\ \mathrm{SpecCtx}(\rho) * j \Rrightarrow K[e'] \twoheadrightarrow \mathsf{wp}\, e \{v.\ \exists v'.\ j \Rrightarrow K[v'] * [\![\Xi \vdash \tau]\!]_{\Delta}(v, v')\}$$

We take $j = 1$, $\rho = (\emptyset, e')$ and $K = []$ and instantiate the above formula to get

$$\mathrm{SpecCtx}(\emptyset, e') * 1 \Rrightarrow e' \twoheadrightarrow \mathsf{wp}\, e \{v.\ \exists v'.\ 1 \Rrightarrow v' * [\![\Xi \vdash \tau]\!]_{\Delta}(v, v')\}$$

Subsequently, we use Iris resource allocation rule to allocate resources for

$$\mathrm{SpecConf}(\emptyset, e') * 1 \Rrightarrow e'$$

(see Appendix D.2 for more details). Now we can establish the invariant corresponding to $\mathrm{SpecCtx}(\emptyset, e')$ by proving

$$\exists \rho'.\ \mathrm{SpecConf}(\rho') \wedge (\emptyset, e') \rightarrow^{*}_{\mathsf{tp}} \rho'$$

We prove this by simply taking $\rho' = (\emptyset, e')$. As a result, we get

$$\mathsf{wp}\, e \{v.\ \exists v'.\ 1 \Rrightarrow v' * [\![\Xi \vdash \tau]\!]_{\Delta}(v, v')\}$$

Now, since we know, by assumption, that $e \downarrow$, we get by the *adequacy* of Iris's weakest preconditions (see Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017)) that any *pure* fact (a fact not depending on step-indices or resources), that follows from the post condition, holds. Hence, from the post condition above, we know, in particular, that there exists a value v' such that $1 \Rrightarrow v'$. Opening the invariant $\mathrm{SpecCtx}(\emptyset, e')$ we get that there is a configuration ρ'' such that $\mathrm{SpecConf}(\rho'') \wedge (\emptyset, e') \rightarrow^{*}_{\mathsf{tp}} \rho''$. Now from $1 \Rrightarrow v'$ and $\mathrm{SpecConf}(\rho'')$ we can conclude, using the rule SPEC-THREAD, that there is a heap σ and a sequence of expressions $\vec{e}$ such that $\rho'' = (\sigma, v'; \vec{e})$. Consequently we have $(\emptyset, e') \rightarrow^{*}_{\mathsf{tp}} (\sigma, v'; \vec{e})$, which is just the definition of $e' \downarrow$. □

We next prove the soundness of our logical relations which simply states that logical refinement implies contextual refinement.

Theorem 5.5.5 (Soundness of binary logical relations)**.**

$$\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau \land \Xi \mid \Gamma \vdash e : \tau \land \Xi \mid \Gamma \vdash e' : \tau \Rightarrow \Xi \mid \Gamma \vDash e \leq_{ctx} e' : \tau$$

Proof. Given a well typed context $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; 1)$ and that

$$C[e] \downarrow$$

we have to prove that $C[e'] \downarrow$. By Lemma 5.5.3 we have that $\cdot \mid \cdot \vDash C[e] \leq_{log} C[e'] : 1$. Unfolding the definition of binary logical relation the latter simplifies to:

$$[\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\epsilon}(C[e], C[e'])$$

Finally, by Lemma 5.5.4 we have $C[e'] \downarrow$, as required. □

Theorem 5.5.5 allows us to prove contextual refinement by showing logical refinement. As our logical relations is formalized on top of the Iris program logic we have the entire power and support of Iris at our disposal when proving logical refinements. In the following two sections we exemplify this by proving contextual refinement of concurrent counters and concurrent stacks.

5.6 Contextual refinement of counters

In this section we establish that a fine-grained implementation of a concurrent counter refines a coarse-grained implementation:

$$\cdot \mid \cdot \vDash \mathbf{Counter}_{\mathsf{Fine}} \leq_{ctx} \mathbf{Counter}_{\mathsf{Coarse}} : (1 \to 1) \times (1 \to \mathbb{N})$$

The two counter modules are closed programs each consisting of a pair of functions. The first function increments the counter and the second reads it. The source code of these two counters is depicted in Figure 5.5. The code in this figure is written in an ML style for conciseness and better readability. The fine-grained counter uses a technique known as optimistic concurrency. It reads the counter and increments it and then tries to set the counter to the incremented value if it has not been changed in the meanwhile. If it fails to do so, it starts over. The coarse-grained counter uses a traditional lock to acquire exclusive access to the counter during the increment operation. Notice that we do not need to acquire the lock to read the counter because reading references is atomic. The lock used by the coarse-grained counter is a spin lock implemented using the `CAS` operation (see Appendix D.3 for details).

```
let Counter_Fine =
  let fc = ref 0 in
  let fincr () =
    let loop () =
      let x = !fc in
      if CAS(fc, x, x+1)
      then ()
      else loop ()
    in
    loop ()
  in
  let fread () = !fc in
  (fincr, fread)
```

```
let Counter_Coarse =
  let cc = ref 0 in
  let lc = new_lock () in
  let cincr () =
    acquire lc;
    cc := !cc + 1;
    release lc
  in
  let cread () = !cc in
  (cincr, cread)
```

Figure 5.5: The source code of fine-grained (left) and coarse-grained (right) concurrent counters

By the soundness of the binary logical relations, Theorem 5.5.5, it suffices to prove the following corresponding logical refinement in order to prove the refinement above.

$$\cdot \mid \cdot \vDash \mathbf{Counter}_{\mathsf{Fine}} \leq_{log} \mathbf{Counter}_{\mathsf{Coarse}} : (1 \to 1) \times (1 \to \mathbb{N})$$

In this section we give sketch the proof of this refinement. The proof consists of two parts. We first need to show that when the fine-grained counter evaluates to a value so does the coarse-grained counter. Afterwards, we need to show that the values on both sides are related. That is, we need to show that the fine-grained increment and read functions are logically related to their coarse-grained counterparts. Hence, we start the first part of the proof by symbolically executing both sides using the rules for weakest preconditions and evaluation on the specification side. The symbolic execution of allocating the counters and the lock gives us the following resources:

$$\texttt{fc} \mapsto 0 \qquad \texttt{cc} \mapsto_s 0 \qquad \texttt{lc} \mapsto_s \texttt{false} \qquad j \Rrightarrow \texttt{(cincr, cread)}$$

while we have to show

$$\mathsf{wp}\ \texttt{(fincr, fread)}\ \{v.\ \exists v'.\ j \Rrightarrow v' * [\![\cdot \vdash (1 \to 1) \times (1 \to \mathbb{N})]\!]_\epsilon(v, v')\}$$

Note that we have omitted the specification side invariant, SpecCtx. Furthermore, the lock that we use is implemented using a single boolean value where `false` indicates that the lock is free (see Appendix D.3 for details).

At this point, we need to show that the increment and read functions of both sides are suitably related. That is, we have to show that the two modules

behave similarly. Hence, there must be a relation between the *state* of the two modules that holds at the beginning and is preserved by every operation of the modules. That is, a relation relating the two counters such that this relation holds at the beginning, i.e., for the resources (the counters and the lock) above. Furthermore, this relation should be preserved throughout the execution of the program including before, during and after each call to increment and read operations. For this proof we use the following relation, expressed in terms of an Iris invariant.

$$\boxed{\exists n.\, \texttt{fc} \mapsto n * \texttt{cc} \mapsto_s n * \texttt{lc} \mapsto_s \texttt{false}}^{\mathcal{N}.Counters} \qquad \text{(Counter-Invariant)}$$

In prose, this invariant states that the two counters always store the same value and the lock of the specification side counter is not acquired. Notice that as far as the behavior of the modules is considered, the lock is never *observed* as acquired. This is because the only acquire statement, in the increment function of the coarse-grained counter, is followed by a release. In proving a refinement we show that if the implementation side converges, then so does the specification side. That is, for the specification side we only need to show that there exists an evaluation that converges to a value.[4] Consequently, we have the discretion to pick the scheduling of the specification side which allows us to ensure that no thread that would attempt to acquire the lock can be executed during the time that the lock is already acquired.

The proof of the refinement for the read functions is straightforward. We open the counter invariant during the atomic read operation of the implementation side where both counters read their corresponding values. The invariant guarantees that the values read by both counters are equal.

The refinement proof of the increment operation boils down to showing that

$$j \Vdash \texttt{cincr}\ () \mathrel{-\!\!*} \texttt{wp}\ \texttt{loop}\ ()\ \{v.\ \exists v'.\ j \Vdash v' * [\![\cdot \vdash 1]\!]_\epsilon(v, v')\}$$

under the assumption that the counter invariant above holds. We proceed by Löb induction which allows us to assume that recursive calls to `loop` satisfy the above weakest preconditions during the proof that the body of the loop satisfies it. The use of the Löb induction principle for recursive functions here is similar to that of the proof of compatibility rule for the typing rule T-REC above. The proof proceeds by reading the fine-grained counter `fc` by opening the counter invariant, (Counter-Invariant), using the rules WP-LOAD and INV-OPEN. Notice that we immediately have to close the invariant after reading `fc`. Let us assume that the value of `fc` that we just read is `n`. Now we need to show (assuming that the addition of $\texttt{n} + 1$ is evaluated to $\texttt{S}(\texttt{n})$ where `S` is the successor function

[4] This form of contextual refinement usually referred to as "*may* contextual refinement" in the literature.

on natural numbers) that the following holds:

$$\mathsf{wp}\ \texttt{if CAS(fc, n, S(n)) then () else loop ()}\ \{v.\ \exists v'.\ j \Rrightarrow v' * [\![\cdot \vdash 1]\!]_\epsilon(v, v')\}$$

under the assumption that $j \Rrightarrow \texttt{cincr ()}$ holds. Using the WP-BIND rule we need to show that

$$\mathsf{wp}\ \texttt{CAS(fc, n, S(n))}\ \left\{ x.\ \begin{array}{l} \mathsf{wp}\ \texttt{if}\ x\ \texttt{then () else loop ()} \\ \{v.\ \exists v'.\ j \Rrightarrow v' * [\![\cdot \vdash 1]\!]_\epsilon(v, v')\} \end{array} \right\}$$

At this point, we can use the rule INV-OPEN to open the counter invariant above. Which allows us to assume that

$$\texttt{fc} \mapsto \texttt{m} * \texttt{cc} \mapsto_s \texttt{m} * \texttt{lc} \mapsto_s \texttt{false}$$

holds for some natural number $\texttt{m}$. Now there are two cases to consider depending on whether $\texttt{n} = \texttt{m}$ holds or not. If $\texttt{n} = \texttt{m}$ then no other thread has incremented the counter between, reading and incrementing the counter, and the CAS operation. In this case, the CAS operation succeeds. Since we have $\texttt{lc} \mapsto_s \texttt{false}$ we know that the acquire operation succeeds giving us $\texttt{lc} \mapsto_s \texttt{true}$. Subsequently, we use the rules for evaluation on the specification side to load, increment and store back the incremented value of the specification side counter, giving us $\texttt{cc} \mapsto_s \texttt{S(n)}$. Finally, we release the lock getting back $\texttt{lc} \mapsto_s \texttt{false}$. This reestablishes the invariant and gives us $j \Rrightarrow ()$ as the release operation returns the () literal value. Thus we have to show that

$$\mathsf{wp}\ \texttt{if true then () else loop ()}\ \{v.\ \exists v'.\ j \Rrightarrow v' * [\![\cdot \vdash 1]\!]_\epsilon(v, v')\}$$

holds. This now holds rather trivially and it is easy to show so.

The remaining case is the case where $\texttt{n} \neq \texttt{m}$. That is, while reading the value of the counter and incrementing it, another thread has managed to successfully increment the counter and has thus invalidated the value that we had read. In this case, the CAS operation fails, resulting in a recursive call. The weakest precondition for the recursive call, however, follows from our Löb induction hypothesis.

5.7 Contextual refinement of stacks

The source code of the two concurrent stack implementations for which we prove refinement are depicted in Figure 5.6. The code in this figure is written in an ML style for conciseness and better readability. Here, on line 1, we use Λ instead of `let` to emphasize that these *values* are polymorphic *values* and are

```
Λ StackFine =
  let fs =
    ref (ref None)
  in
  let fpush x =
    let pushloop () =
      let z = fold !fs in
      let w =
        (ref (Some (x, z)))
      in
      if CAS(fs, z, w) then
        ()
      else
        pushloop ()
    in
    pushloop ()
  in
  let fpop x =
    let poploop () =
      let z = !fs in
      match !z with
      | None -> None
      | Some (top, rest) ->
         let w = unfold rest
         in
         if CAS(fs, z, w)
         then Some top
         else poploop ()
    in
    poploop ()
  in
  let fiter f =
    iloop v =
      match unfold v with
      | None -> ()
      | Some (top, rest) ->
          f top; iloop rest
    in iloop (fold !fs)
  let fread () = !fc in
  (fpush, fpop, fiter)
```

```
Λ StackCoarse =
  let cs = ref [] in
  let lc = new_lock ()
  in
  let cpush x =
    acquire lc;
    cs := x :: !cs;
    release lc
  in
  let cpop () =
    acquire lc;
    let z =
      match !cs with
      | [] -> None
      | hd :: tl ->
          cs := tl; hd
    in
    release lc; z
  in
  let citer f =
    let iloop v =
      match v with
      | [] -> ()
      | hd :: tl ->
          f hd;
          iloop tl
    in
    iloop !cs
  in
  (cpush, cpop, citer)
```

Figure 5.6: The source code of fine-grained (left) and coarse-grained (right) concurrent stacks

not evaluated until they are applied to a type. Lists and the option type are encoded in the usual way, the former using recursive types:

$$\tau\ \texttt{list} \triangleq \mu X.\ (\tau \times X)\ \texttt{option} \qquad \tau\ \texttt{option} \triangleq 1 + \tau$$

$$[] \triangleq \texttt{fold None} \qquad \texttt{None} \triangleq \texttt{inj}_1\,()$$

$$\texttt{hd} :: \texttt{tl} \triangleq \texttt{fold}\ (\texttt{Some}\ (hd, tl)) \qquad \texttt{Some}\ v \triangleq \texttt{inj}_2\,v$$

The type of both of these implementations is

$$\texttt{StackT} \triangleq \forall X.\,(X \to 1) \times (1 \to X\ \texttt{option}) \times ((X \to 1) \to 1)$$

These modules provide three functions for three operations: pushing, popping and iterating over the elements of the stack. The coarse grained stack simply stores the entire stack as a list in a reference. It locks the whole data structure during pushing and popping values. The iterator function, given a function to apply to elements on the stack, reads the stack atomically and applies this function to every element of the list representing the stack.

The fine-grained stack uses a linked-list implementation to represent the stack. The type of the cells of linked-list is as follows:

$$\tau\ \texttt{cell} \triangleq \mu X.\,\texttt{ref}((\tau \times X)\ \texttt{option})$$

where τ is the type of the elements stored on the stack. That is, each cell is a reference whose content, if available, is a value together with a reference to another cell, which represents the next pointer of the linked-list. The stack has a head pointer, which has the type $\texttt{ref}((\tau \times \tau\ \texttt{cell})\ \texttt{option})$.[5] The push and pop operations only change the head pointer of the stack. In other words, the cells are immutable in practice in the sense that we never change any of the values stored in the references in cells. For this reason, once we read the head pointer we know that the linked-list obtained will never change even though the stack can change. This is crucial for the refinement of iterators.

In this section we will discuss the following refinement:

$$\cdot \mid \cdot \vDash \mathbf{Stack}_{\mathsf{Fine}} \leq_{ctx} \mathbf{Stack}_{\mathsf{Coarse}} : \texttt{StackT}$$

The proof of refinements for individual operations of the stack modules are straightforward but lengthy. Hence, we omit these here. Suffice it to say that CAS loops in the push and pop functions of the fine-grained implementation are treated similarly to the increment function of the counter refinement proof.

[5] We store the unfolded version of the head cell as we cannot perform CAS on references whose contents are recursive types. See the typing rule T-CAS for details.

Furthermore, the iteration operations are almost identical, they both read the entire stack (the fine-grained version reads the head pointer which points to a first cell of the stack) and loop over the contents applying each their given function. What is crucial in this proof is the invariant that relates the internal representation of the two modules. This is what we will discuss in what remains of this section.

Reasoning about stack cells As mentioned earlier, cells of linked-lists used to represent fine-grained stacks are immutable in the sense that these are references whose content never changes. This is crucial for correctness of the iterator and the relatedness of its two different implementations. Here, we introduce Iris propositions for reasoning about these immutable cells. These propositions are encoded using Iris resources. Details of the encoding can be found in Appendix D.4. The propositions that we introduce are $\mathrm{AllCells}(f)$ and $\ell \mapsto^{stk} v$, where $f : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$ is a finite partial map from locations to values. Intuitively, the proposition $\mathrm{AllCells}(f)$ holds if all the cells ever used in the stack, i.e., cells that currently are or previously have been part of the stack, are described by f. The proposition $\ell \mapsto^{stk} v$ states that there is a stack cell at location l with value v. The following rules govern propositions $\mathrm{AllCells}(f)$ and $\ell \mapsto^{stk} v$:

CELL-PERSISTENT
$$\mathsf{persistent}(\ell \mapsto^{stk} v)$$

CELL-AGREE
$$\frac{\ell \mapsto^{stk} v \qquad \ell \mapsto^{stk} w}{v = w}$$

CELL-IN-ALL
$$\frac{\mathrm{AllCells}(f) \qquad \ell \mapsto^{stk} v}{f(\ell) = v}$$

CELL-ALLOC
$$\frac{\mathrm{AllCells}(f) \qquad \ell \notin \mathrm{dom}(f)}{\Rrightarrow_{\mathcal{E}} \mathrm{AllCells}(f[\ell \mapsto v]) * \ell \mapsto^{stk} v}$$

ALL-CELLS-UNIQUE
$$\mathrm{AllCells}(f) * \mathrm{AllCells}(g) \vdash \bot$$

Notice that immutability of cells is guaranteed by the rules CELL-PERSISTENT and CELL-AGREE.

To associate the introduced Iris propositions above to the physical stack cells in the memory we define the following Iris predicates:

$$\mathrm{StackOwns}(f) \triangleq \mathrm{AllCells}(f) * \mathop{\Large *}_{(\ell,v)\in f} \ell \mapsto v$$

$$\mathrm{LRel}_{\Phi}(\ell, v') \triangleq \mu r.\ (\ell \mapsto^{stk} \texttt{None} * v' = [])\vee$$
$$\exists w, w', u', m.\ \ell \mapsto^{stk} \texttt{Some}\ (w, \texttt{fold}\, m) * v' = w' :: u' *$$
$$\Phi(w, w') * \triangleright\, r(m, u')$$

The predicate $\mathrm{StackOwns}(f)$ asserts that all of the cells are described by f and that we own the memory corresponding to all of these cells. The relation $\mathrm{LRel}_{\Phi}(\ell, v)$ ensures that the linked-list pointed by ℓ and the list v have the same length and that corresponding values in these lists are related by the value relation Φ. For these predicates we can derive the following rules:

$$\textsc{stack-reveal}\quad \frac{\mathrm{StackOwns}(f) \qquad \ell \mapsto^{stk} v}{\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \mathrm{StackOwns}(f))} \qquad \textsc{stack-alloc}\quad \frac{\mathrm{StackOwns}(f) \qquad \ell \mapsto v}{\Rrightarrow_{\mathcal{E}} \mathrm{StackOwns}(f[\ell \mapsto v]) * \ell \mapsto^{stk} v}$$

$$\textsc{LRel-persistent}\quad \frac{\mathsf{persistent}(\Phi)}{\mathsf{persistent}(\mathrm{LRel}_{\Phi}(\ell, v))}$$

The stack invariant The two stack implementations that we are considering are polymorphic. Hence, in proving their refinement we have to maintain that the contents stored in the two stacks are related by the binary value relation corresponding to the types that stacks are applied to. Let us fix $\Phi_{stk} : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ as this binary value relation. The invariant relating the internal representation of the two stacks is defined as follows:

$$\boxed{\exists f, \ell, v'.\ \mathtt{fc} \mapsto \mathtt{fold}\,\ell * \mathtt{cc} \mapsto_s v' * \mathtt{lc} \mapsto_s \mathtt{false} \\ * \mathrm{StackOwns}(f) * \mathrm{LRel}_{\Phi_{stk}}(\ell, v')}^{\mathcal{N}.\mathit{Stacks}} \qquad \text{(Stack-Invariant)}$$

5.8 Related work

The idea of formalizing the logical relations for a programming language on top of a logic goes back to Plotkin and Abadi (1993) who to specify such a logical relation for System F and Dreyer, Ahmed, and Birkedal (2009) who constructed such a logical relation for a programming language featuring recursive types. Our Logical relations model is inspired by the model of Krogh-Jespersen, Svendsen, and Birkedal (2017) which is in turn based on the model by Turon, Dreyer, and Birkedal (2013).

There has been a lot of earlier work on concrete logical relations models (not formalized in some specialized logic) for reasoning about contextual refinement. Kripke logical relations models for higher-order languages with dynamically allocated state include Ahmed, Dreyer, and Rossberg, 2009; Birkedal, Reus, Schwinghammer, Støvring, Thamsborg, and Yang, 2011; Birkedal, Støvring, and Thamsborg, 2009; Dreyer, Neis, and Birkedal, 2010. The first logical relation

for a higher order language similar to the one considered here was developed in Birkedal, Sieczkowski, and Thamsborg, 2012, but it only used a simple kind of Kripke world, which did not suffice for reasoning about fine-grained concurrent data structures. The model in Turon, Thamsborg, Ahmed, Birkedal, and Dreyer, 2013 improved on Birkedal, Sieczkowski, and Thamsborg, 2012 by using more sophisticated worlds, which allowed to reason about fine-grained concurrent data structures, and it formed the basis for the logical treatment in Turon, Dreyer, and Birkedal, 2013 mentioned above.

5.9 Conclusion

Type soundness and contextual refinements are two of the most important problems in the study of programs and programming languages. In this paper we presented operationally-based logical relations models formalized in the Iris program logic to tackle these problems. In particular, we have developed our logical relations models for $\mathsf{F}_{\mu,ref,conc}$, a programming language featuring polymorphism, recursive types, higher-order references and concurrency. We used our unary logical relations model to prove type soundness of $\mathsf{F}_{\mu,ref,conc}$. We used our binary logical relations model to prove that fine-grained implementations of concurrent counters and stacks refine their coarse-grained counterparts.

It is well-known that developing and using logical relations models for programming languages with advanced type systems, e.g., featuring both polymorphism and higher-order references, is intricate. Using the logical framework of the Iris program logic allowed us to avoid these intricacies by reasoning about them at a high level of abstraction. Another advantage of working in Iris is that we can use its rich logic, e.g., invariants, resources, weakest preconditions, etc., to prove contextual refinement of programs. We illustrated this in proving refinements for concurrent counter and stack modules. Finally, working in Iris allowed us to use Iris as a Coq library to facilitate formalizing our results in Coq. This is the first time that logical relations models are formalized in a proof assistant for a programming language as expressive as $\mathsf{F}_{\mu,ref,conc}$.

Chapter 6

A Logical Relation for Monadic Encapsulation of State

Proving Contextual Equivalences in the Presence of RunST

This chapter and its appendix are published in the proceedings of the 45^{th} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18) (Timany, Stefanesco, Krogh-Jespersen, and Birkedal, 2018).

We present a logical relations model of a higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with `runST`. We use our logical relations model to show that `runST` provides proper encapsulation of state, by showing that effectful computations encapsulated by `runST` are heap independent. Furthermore, we show that contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of `runST`. This is the first time such relational results have been proven for a language with monadic encapsulation of state. We have formalized all the technical development and results in Coq.

6.1 Introduction

Haskell is often considered a *pure* functional programming language because effectful computations are encapsulated using monads. To preserve purity, values usually cannot escape from those monads. One notable exception is the ST monad, introduced by Launchbury and Peyton Jones (1994). The ST monad comes equipped with a function $\texttt{runST} : (\forall \beta, \texttt{ST}\ \beta\ \tau) \to \tau$ that allows a value to escape from the monad: `runST` runs a stateful computation of the monadic type $\texttt{ST}\ \beta\ \tau$ and then returns the resulting value of type τ. In the original paper Launchbury and Peyton Jones (1994), the authors argued informally that the ST monad is "safe", in the sense that stateful computations are properly encapsulated and therefore the purity of the functional language is preserved.

In this paper we present a logical relations model of STLang, a call-by-value higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with `runST`. In contrast to earlier work, the operational semantics of STLang uses a *single global mutable heap*, capturing how the language would be implemented in reality. We use our logical relations model to show for the first time that `runST` provides proper encapsulation of state. Concretely, we state a number of contextual refinements and equivalences that are expected to hold for pure computations and we then use our logical relations model to prove that they indeed hold for STLang, i.e., in the presence of stateful computations encapsulated using `runST`. Moreover, we show a State-Independence theorem that intuitively expresses that, for any well-typed expression e of type τ, the evaluation of e in a heap h is independent of the choice of h, i.e., e cannot read from or write to locations in h but may allocate new locations (via encapsulated stateful computations). Note that this is the strong result one really wishes to have since it is proved for a standard operational semantics using a single global mutable heap allowing for updates in-place, not an abstract semantics partitioning memory into disjoint

regions as some earlier work (Launchbury and Peyton Jones, 1995; Moggi and Sabry, 2001).

In STLang, values of any type can be stored in the heap, and thus it is an example of a language with so-called higher-order store. It is well-known that it is challenging to construct logical relations for languages with higher-order store. We define our logical relations model in Iris, a state-of-the-art higher-order separation logic (Jung, Krebbers, Birkedal, and Dreyer, 2016; Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer, 2015; Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). Iris's base logic (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017) comes equipped with certain modalities which we use to simplify the construction of the logical relation. Logical relations for other type systems have recently been defined in Iris (Krebbers, Timany, and Birkedal, 2017; Krogh-Jespersen, Svendsen, and Birkedal, 2017), but to make our logical relations model powerful enough to prove the contextual equivalences for purity, we use a new approach to defining logical relations in Iris, which involves several new technical innovations, described in §6.3.

Another reason for using Iris is that the newly developed powerful proof mode for Iris (Krebbers, Timany, and Birkedal, 2017) makes it possible to conduct interactive proofs in the Iris logic in Coq, much in the same way as one normally reasons in the Coq logic itself. Indeed, we have used the Iris proof mode to formalize all the technical results in this paper in Iris in Coq.

In the remainder of this Introduction, we briefly recap the Haskell ST monad and recall why `runST` intuitively encapsulates state. We emphasize that STLang, unlike Haskell, is call-by-value; we show Haskell code to make the examples easier to understand. Finally, we give an overview of the technical development and our new results.

6.1.1 A Recap of the Haskell ST Monad

The ST monad, as described in (Launchbury and Peyton Jones, 1994) and implemented in the standard Haskell library, is actually a family $ST\ \beta$ of monads, where β ranges over types, which satisfy the following interface. The first two functions

```
return :: α → ST β α
(>>=)  :: ST β α → (α → ST β α') → ST β α'
```

are the standard Kleisli arrow interface of monads in Haskell; `>>=` is pronounced "bind". Recall that in Haskell, free type variables (α, α', and β above) are

implicitly universally quantified.[1]

The next three functions

```
newSTRef   :: α → ST β (STRef β α)
readSTRef  :: STRef β α → ST β α
writeSTRef :: STRef β α → α → ST β ()
```

are used to *create*, *read from* and *write into* references, respectively. Notice that the reference type `STRef` β τ, contains the type of the contents of the reference cells, τ, but also another type parameter, β, which, intuitively, indicates which (logical) region of the heap this reference belongs to. The interesting part of the interface is the interaction of this type parameter with the following function

```
runST :: (∀ β. ST β α) → α
```

The `runST` function runs effectful computations and extracts the result from the ST monad. Notice the impredicative quantification of the type variable of `runST`.

Finally, equality on references is decidable:

```
(==) :: STRef β α → STRef β α → bool
```

Notice that equality is an ordinary function, since it returns a boolean value directly, not a value of type `ST` β `bool`.

Figure 6.1 shows how to compute the n-th term of the Fibonacci sequence in Haskell using the ST monad and, for comparison, in our model language STLang. Haskell programmers will notice that the STLang program on the right is essentially the same as the one on the left after the do-notation has been expanded. The inner function `fibST'` can be typed as follows:

```
fibST' :: Integer → STRef β Integer → STRef β Integer →
  ST β Integer
```

Hence, the argument of `runST` has type `(∀ β. STRef β Integer)` and thus `fibST` indeed has return type `Integer`.

[1] In STLang, we use capital letters, e.g. X, for type variables and use ρ for the index type in `ST` ρ τ and `STRef` ρ τ.

```
fibST :: Integer → Integer
fibST n =
 let fibST' 0 x _ =
       readSTRef x
     fibST' n x y = do
       x' <- readSTRef x
       y' <- readSTRef y
       writeSTRef x y'
       writeSTRef y (x'+y')
       fibST' (n-1) x y
 in
 if n < 2 then n else
  runST $ do
    x <- newSTRef 0
    y <- newSTRef 1
    fibST' n x y
```

```
let fibST : ℤ -> ℤ =
 let rec fibST' n x y =
  if n = 0 then !x
   else
    bind !x in λ x' ->
    bind !y in λ y' ->
    bind x := y' in λ () ->
    bind y := (x'+y') in
    λ () ->
     fibST' (n-1) x y
  in
  if n < 2 then n else
   runST {
    bind (ref 0) in λ x ->
     bind (ref 1) in λ y ->
      fibST' n x y }
```

Figure 6.1: Computing Fibonacci numbers using the ST monad in Haskell (left) and in STLang (right). Haskell code adapted from https://wiki.haskell.org/Monad/ST. do is syntactic sugar for wrapping bind around a sequence of expressions.

6.1.2 Encapsulation of State Using `runST`: What is the Challenge?

The operational semantics of the `newSTRef`, `readSTRef`, `writeSTRef` operations is intended to be the same as for ML-style references. In particular, an implementation should be able to use a global heap and in-place update for the stateful operations. The ingenious idea of Launchbury and Peyton Jones (1994) is that the parametric polymorphism in the type for `runST` should still ensure that stateful computations are properly encapsulated and thus that ordinary functions remain pure.

The intuition behind this intended property is that the first type variable parameter of ST, denoted β above, actually denotes a region of the heap, and that we can *imagine* that the heap consists of a collection of disjoint regions, named by types. A computation e of type $ST\ \beta\ \tau$ can then read, write, and allocate in the region named β, and then produce a value of type τ.

Moreover, if e has type $\forall\beta.\ ST\ \beta\ \tau$, with β not free in τ, the intuition is that `runST` e can allocate a fresh region, which e may use and then, since β is not free in τ, the resulting value of type τ cannot involve references in the region β.

It is therefore safe to discard the region β and return the value of type τ. Since stateful computations intuitively are encapsulated in this way, this should also entail that the rest of the "pure" language indeed remains pure. For example, it should still be the case that for an expression e of type τ, running e twice should be the same as running it once. More precisely, we would expect the following contextual equivalence to hold for any expression e of type τ:

$$\texttt{let}\, x = e\, \texttt{in}\, (x, x) \quad \approx_{ctx} \quad (e, e) \qquad (*)$$

Note that, of course, this contextual equivalence would not hold in the presence of unrestricted side effects as in ML: if `e` is the expression `y := !y + 1`, which increments the reference `y`, then the reference would be incremented by 1 on the left hand-side of $(*)$ and by 2 on the right.

Similar kinds of contextual equivalences and refinements that we expect should hold for a pure language should also continue to hold. Moreover, we also expect that the State-Independence theorem described above should hold.

Notice that this intuitive explanation is just a conceptual model — *the real implementation of the language uses a standard global heap with in-place update and the* challenge *is to prove that the type system still enforces this intended proper encapsulation of effects.*

In this paper, we provide a solution to this challenge: we define a higher-order functional programming language, called STLang, with impredicative polymorphism, recursive types, and a Haskell-style *ST* monad type with `runST`. The operational semantics uses a global mutable heap for stateful operations. We develop a logical relations model which we use to prove contextual refinements and equivalences that one expects should hold for a pure language in the presence of stateful computations encapsulated using `runST`.

Earlier work has focused on *simpler* variations of this challenge; specifically, it has focused on type safety, and none of the earlier formal models can be used to show expected contextual equivalences for the pure part of the language relative to an operational semantics with a single global mutable heap. In particular, the semantics and parametricity results of Launchbury and Peyton Jones (1995) is denotational and does not use a global mutable heap with in-place update, and they state Launchbury and Peyton Jones, 1995, Section 9.1 that proving that the remaining part of the language remains pure for an implementation with in-place update "would necessarily involve some operational semantics." We discuss other related work in §6.5.

6.1.3 Overview of Results and the Technical Development

In §6.2 we present the operational semantics and the type system for our language STLang. In this paper, we focus on the encapsulation properties of a Haskell-style monadic type system for stateful computations. The choice of evaluation order is an orthogonal issue and, for simplicity (to avoid having to formalize a lazy operational semantics), we use call-by-value left-to-right evaluation order. Typing judgments take the standard form $\Xi \mid \Gamma \vdash e : \tau$, where Ξ is an environment of type variables, Γ an environment associating types to variables, e is an expression, and τ is a type. For well-typed expressions e and e' we define contextual refinement, denoted $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau$. As usual, e and e' are contextually equivalent, denoted $\Xi \mid \Gamma \vDash e \approx_{ctx} e' : \tau$, if e contextually refines e' and vice versa. With this in place, we can explain which contextual refinements and equivalences we prove for STLang. The soundness of these refinements and equivalences means, of course, that one can use them when reasoning about program equivalences.

The contextual refinements and equivalences that we prove for pure computations are given in Figure 6.2. To simplify the notation, we have omitted environments Ξ and Γ in the refinements and equivalences in the Figure. Moreover, we do not include assumptions on typing of subexpressions in the Figure; precise formal results are stated in §6.4.

Refinement (Neutrality) expresses that a computation of unit type either diverges or produces the unit value. The contextual equivalence in (Commutativity) says that the order of evaluation for pure computations does not matter: the computation on the left first evaluates e_2 and then e_1, on the right we first evaluate e_1 and then e_2. The contextual equivalence in (Idempotency) expresses the idempotency of pure computations: it does not matter whether we evaluate an expression once, as done on the left, or twice, as done on the right. The contextual refinements in (Rec hoisting) and (Λ hoisting) formulate the soundness of λ-hoisting for ordinary recursive functions and for type functions. The contextual refinements (η expansion for rec) and (η expansion for Λ) express η-rules for ordinary recursive functions and for type functions. The contextual refinements (β reduction for rec) and (β reduction for Λ) express the soundness of β-rules for ordinary recursive functions and for type functions.

In addition, we prove the expected contextual equivalences for monadic computations, shown in Figure 6.3.

The results in Figure 6.2 are the kind of results one would expect for pure computations in a call-by-value language; the challenge is, of course, to show that they hold in the full STLang language, that is, also when subexpressions

$$e \preceq_{\mathsf{ctx}} () : 1 \qquad \text{(Neutrality)}$$

$$\mathtt{let}\, x = e_2\, \mathtt{in}\, (e_1, x) \approx_{ctx} (e_1, e_2) : \tau_1 \times \tau_2 \qquad \text{(Commutativity)}$$

$$\mathtt{let}\, x = e\, \mathtt{in}\, (x, x) \approx_{ctx} (e, e) : \tau \times \tau \qquad \text{(Idempotency)}$$

$$\mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{rec}\, f(x) = e_2 \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = \mathtt{let}\, y = e_1\, \mathtt{in}\, e_2 : \tau_1 \to \tau_2 \qquad \text{(Rec hoisting)}$$

$$\mathtt{let}\, y = e_1\, \mathtt{in}\, \Lambda\, e_2 \preceq_{\mathsf{ctx}} \Lambda\, (\mathtt{let}\, y = e_1\, \mathtt{in}\, e_2) : \forall X.\, \tau \quad (\Lambda \text{ hoisting})$$

$$e \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = (e\ x) : \tau_1 \to \tau_2 \qquad (\eta \text{ expansion for rec})$$

$$e \preceq_{\mathsf{ctx}} \Lambda\, (e\ \_) : \forall X.\, \tau \qquad (\eta \text{ expansion for } \Lambda)$$

$$(\mathtt{rec}\, f(x) = e_1)\ e_2 \preceq_{\mathsf{ctx}} e_1[e_2, (\mathtt{rec}\, f(x) = e_1)/x, f] : \tau \qquad (\beta \text{ reduction for rec})$$

$$(\Lambda\, e)\ \_ \approx_{ctx} e : \tau[\tau'/X] \qquad (\beta \text{ reduction for } \Lambda)$$

Figure 6.2: Contextual Refinements and Equivalences for Pure Computations

$$\mathtt{bind}\, e\, \mathtt{in}\, (\lambda\, x.\, \mathtt{return}\, x) \approx_{ctx} e : \mathtt{ST}\ \rho\ \tau \qquad \text{(Left Identity)}$$

$$e_2\ e_1 \preceq_{\mathsf{ctx}} \mathtt{bind}\, (\mathtt{return}\, e_1)\, \mathtt{in}\, e_2 : \mathtt{ST}\ \rho\ \tau \qquad \text{(Right Identity)}$$

$$\mathtt{bind}\, (\mathtt{bind}\, e_1\, \mathtt{in}\, e_2)\, \mathtt{in}\, e_3 \preceq_{\mathsf{ctx}} \mathtt{bind}\, e_1\, \mathtt{in}\, (\lambda\, x.\, \mathtt{bind}\, (e_2\ x)\, \mathtt{in}\, e_3) : \mathtt{ST}\ \rho\ \tau' \qquad \text{(Associativity)}$$

Figure 6.3: Contextual Equivalences for Stateful Computations

may involve arbitrary (possibly nested) stateful computations encapsulated using `runST`. That is the purpose of our logical relation, which we present in §6.3. We further use our logical relation to show the following State-Independence theorem:

Theorem 6.1.1 (State Independence)**.**

$$\cdot \mid x : \texttt{STRef}\ \rho\ \tau' \vdash e : \tau \wedge (\exists h_1, \ell, h_2, v.\ \langle h_1, e[\ell/x]\rangle \to^* \langle h_2, v\rangle) \implies$$
$$\forall h_1', \ell'.\ \exists h_2', v'.\ \langle h_1', e[\ell'/x]\rangle \to^* \langle h_2', v'\rangle \wedge h_1' \subseteq h_2'.$$

This theorem expresses that, if the execution of a well-typed expression e, when x is substituted by some location, in *some* heap h_1 terminates, then running e, when x is substituted by *any* location, in *any* heap h_1' will also terminate in some heap h_2' which is an extension of h_1', i.e., the execution cannot have modified h_1' but it can have allocated new state, via encapsulated stateful computations. Note that this implies that e never reads from or writes to x.

Summary of contributions To sum up, the main contributions of this paper are as follows:

- We present a logical relation for a programming language STLang featuring a parallel to Haskell's ST monad with a construct, `runST`, to encapsulate stateful computations. We use our logical relation to prove that `runST` provides proper encapsulation of state, by showing (1) that contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of stateful computations encapsulated via `runST` and (2) that the State-Independence theorem holds. This is the first time that these results have been established for a programming language with an operational semantics that uses a single global higher-order heap with in-place destructive updates.
- We define our logical relation in Iris, a state-of-the-art higher-order separation logic designed for program verification, using a new approach involving novel predicates defined in Iris, which we explain in §6.3.
- We have formalized the whole technical development, including all proofs of the equations above and the State-Independence theorem, in the Iris implementation in Coq.

The paper is organized as follows. We begin by formally defining STLang, its semantics and typing rules in §6.2. There, we also formally state our definition of contextual refinement and contextual equivalence. In §6.3, we present our

$$\begin{aligned}
\circledcirc &::= + \mid - \mid * \mid = \mid < \\
e &::= x \mid () \mid \texttt{true} \mid \texttt{false} \mid n \mid \ell \mid (e, e) \mid \texttt{inj}_i\, e \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, e \\
&\quad \mid \texttt{unfold}\, e \mid e\ e \mid e\ _ \mid \pi_i\, e \mid \texttt{match}\, e\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} \\
&\quad \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid e \circledcirc e \mid \texttt{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid e == e \mid \texttt{bind}\, e\, \texttt{in}\, e \\
&\quad \mid \texttt{return}\, e \mid \texttt{runST}\ \{e\} \\
v &::= () \mid \texttt{true} \mid \texttt{false} \mid n \mid \ell \mid (v, v) \mid \texttt{inj}_i\, v \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, v \\
&\quad \mid \texttt{ref}(v) \mid\ !\, v \mid v \leftarrow v \mid \texttt{bind}\, v\, \texttt{in}\, v \mid \texttt{return}\, v \\
\tau &::= X \mid \rho \mid 1 \mid \mathbb{B} \mid \mathbb{Z} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall X.\, \tau \mid \mu X.\, \tau \mid \texttt{ref}(\tau) \mid \texttt{ST}\ \rho\ \tau
\end{aligned}$$

Figure 6.4: The syntax of STLang

logical relation after briefly introducing the parts of Iris needed for a conceptual understanding of the logical relation. We devote §6.4 to the precise statement and proof sketches of the refinements in Figure 6.2 and Figure 6.3. We discuss related work in §6.5 and conclude in §6.6.

6.2 The STLang language

In this section, we present STLang, a higher-order functional programming language with impredicative polymorphism, recursive types, higher-order store and a ST-like type.

Syntax The syntax of STLang is mostly standard and presented in Figure 6.4. Note that there are no types in the terms; following Ahmed (2006) we write $\Lambda\, e$ for type abstraction and $e\ \_$ for type application / instantiation. For the stateful part of the language, we use `return` and `bind` for the return and bind operations of the ST monad, and `ref`(e) creates a new reference, `!e` reads from one and `e` $\leftarrow$ `e` writes into one. Finally, `runST` runs effectful computations. Note that we treat the stateful operations as constructs in the language rather than as special constants.

$\boxed{\Xi \mid \Gamma \vdash e : \tau}$

$$\textsc{Tvar}\ \frac{}{\Xi \mid \Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Trec}\ \frac{\Xi \mid \Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2}{\Xi \mid \Gamma \vdash \texttt{rec}\, f(x) = e : \tau_1 \to \tau_2} \qquad \textsc{Tabs}\ \frac{\Xi, X \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall X.\, \tau}$$

$$\textsc{Tfold}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau / X]}{\Xi \mid \Gamma \vdash \texttt{fold}\, e : \mu X.\, \tau} \qquad \textsc{Tinst}\ \frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]}$$

$$\textsc{Tnew}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \texttt{ref}(e) : \texttt{ST}\ \rho\ (\texttt{STRef}\ \rho\ \tau)} \qquad \textsc{Tderef}\ \frac{\Xi \mid \Gamma \vdash e : \texttt{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash\, !\, e : \texttt{ST}\ \rho\ \tau}$$

$$\textsc{Tgets}\ \frac{\Xi \mid \Gamma \vdash e : \texttt{STRef}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : \texttt{ST}\ \rho\ 1} \qquad \textsc{Trunst}\ \frac{\Xi, X \mid \Gamma \vdash e : \texttt{ST}\ X\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \texttt{runST}\ \{e\} : \tau}$$

$$\textsc{Trefeq}\ \frac{\Xi \mid \Gamma \vdash e : \texttt{STRef}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \texttt{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash e == e' : \mathbb{B}} \qquad \textsc{Treturn}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \texttt{return}\, e : \texttt{ST}\ \rho\ \tau}$$

$$\textsc{Tbind}\ \frac{\Xi \mid \Gamma \vdash e : \texttt{ST}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau \to (\texttt{ST}\ \rho\ \tau')}{\Xi \mid \Gamma \vdash \texttt{bind}\, e\, \texttt{in}\, e' : \texttt{ST}\ \rho\ \tau'}$$

Figure 6.5: An excerpt of the typing rules for STLang

Typing Typing judgments are of the form $\Xi \mid \Gamma \vdash e : \tau$, where Ξ is a set of type variables, and Γ is a finite partial function from variables to types. An excerpt of the typing rules are shown in Figure 6.5.

Operational semantics We present a small-step call-by-value operational semantics for STLang, using a transition system $\langle h, e\rangle \to \langle h', e'\rangle$ whose nodes are configurations consisting of a heap h and an expression e. A heap $h \in \mathit{Loc} \rightharpoonup^{\text{fin}} \mathit{Val}$ is a finite partial function that associates values to locations, which we suppose are positive integers ($\mathit{Loc} \triangleq \mathbb{Z}^+$)[2].

[2]This choice is due to the fact that Iris library in Coq provides extensive support for the type of positive integers.

Reduction: $\boxed{\langle h, e\rangle \to \langle h', e'\rangle}$ and head step: $\boxed{\langle h, e\rangle \to_h \langle h', e'\rangle}$
Evaluation contexts:

$$K ::= [] \mid (K, e) \mid (v, K) \mid \mathtt{inj}_i\, K \mid \mathtt{fold}\, K \mid \mathtt{unfold}\, K \mid K\ e \mid v\ K \mid K\ \_ \mid$$
$$K \circledcirc e \mid v \circledcirc K \mid \pi_i\, K \mid \mathtt{match}\, K\, \mathtt{with}\, \mathtt{inj}_i\, x \Rightarrow e_i\, \mathtt{end} \mid$$
$$\mathtt{if}\, K\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{ref}(K) \mid\ !K \mid K \leftarrow e \mid v \leftarrow K \mid K == e \mid$$
$$v == K \mid \mathtt{bind}\, K\, \mathtt{in}\, e \mid \mathtt{bind}\, v\, \mathtt{in}\, K \mid \mathtt{return}\, K \mid \mathtt{runST}\ \{K\}$$

$$\frac{\langle h, e\rangle \to_h \langle h', e'\rangle}{\langle h, K[e]\rangle \to \langle h', K[e']\rangle} \qquad \langle h, \mathtt{unfold}\,(\mathtt{fold}\, v)\rangle \to_h \langle h, v\rangle$$

$$\langle h, (\Lambda\, e)\ \_\rangle \to_h \langle h, e\rangle \qquad \langle h, (\mathtt{rec}\, f(x) = e)\ v\rangle \to_h \langle h, e[v, \mathtt{rec}\, f(x) = e/x, f]\rangle$$

$$\frac{\ell = \ell'}{\langle h, \ell == \ell'\rangle \to_h \langle h, \mathtt{true}\rangle}$$

$$\langle h, \mathtt{match}\, \mathtt{inj}_i\, v\, \mathtt{with}\, \mathtt{inj}_i\, x \Rightarrow e_i\, \mathtt{end}\rangle \to_h \langle h, e_i[v/x]\rangle$$

$$\frac{\ell \neq \ell'}{\langle h, \ell == \ell'\rangle \to_h \langle h, \mathtt{false}\rangle} \qquad \frac{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}{\langle h, \mathtt{runST}\ \{v\}\rangle \to_h \langle h', \mathtt{runST}\ \{e\}\rangle}$$

$$\langle h, \mathtt{runST}\ \{\mathtt{return}\, v\}\rangle \to_h \langle h, v\rangle$$

Figure 6.6: An excerpt of the dynamics of STLang, a call-by-value, small-step operational semantics (part 1)

The semantics, shown in Figures 6.6 and 6.7, is presented in the Felleisen-Hieb style (Felleisen and Hieb, 1992), using evaluation contexts K: the reduction relation $\to$ is the closure by evaluation context of the *head* reduction relation $\to_h$. Notice that even the "pure" reductions steps, such as β-reduction, mention the heap. The more subtle part of the operational semantics is how the ST monad is handled, indeed, we only want the stateful computations to run when they are wrapped inside `runST`. This is why we define an auxiliary reduction relation, $\langle h, e\rangle \rightsquigarrow \langle h', e'\rangle$. This auxiliary relation is also defined using a head reduction and evaluation contexts $\mathbb{K}$, which are distinct from the evaluation contexts for the main reduction relation. This auxiliary relation in "embedded" in the main

Effectful reduction: $\boxed{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}$ and effectful head step: $\boxed{\langle h, v\rangle \rightsquigarrow_h \langle h', e\rangle}$

Effectful evaluation contexts: $\mathbb{K} ::= [] \mid \texttt{bind}\,\mathbb{K}\,\texttt{in}\,v$

$$\frac{\langle h, v\rangle \rightsquigarrow_h \langle h', e\rangle}{\langle h, \mathbb{K}[v]\rangle \rightsquigarrow \langle h', \mathbb{K}[e]\rangle} \qquad \langle h, \texttt{bind}\,(\texttt{return}\,v)\,\texttt{in}\,v'\rangle \rightsquigarrow_h \langle h, v'\ v\rangle$$

$$\textsc{Alloc}\ \frac{\ell \notin \mathrm{dom}(h)}{\langle h, \texttt{ref}(v)\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \texttt{return}\,\ell\rangle}$$

$$\langle h \uplus \{\ell \mapsto v\}, !\,\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \texttt{return}\,v\rangle$$

$$\langle h \uplus \{\ell \mapsto v'\}, \ell \leftarrow v\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \texttt{return}\,()\rangle$$

If $\rightharpoonup$ is a relation, we note $\rightharpoonup^n$ its iterated self-composition and $\rightharpoonup^*$ its reflexive and transitive closure.

Figure 6.7: An excerpt of the dynamics of STLang, a call-by-value, small-step operational semantics (part 2)

one by the rule

$$\frac{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}{\langle h, \texttt{runST}\ \{v\}\rangle \rightarrow_h \langle h', \texttt{runST}\ \{e\}\rangle}$$

Notice that $\rightsquigarrow$ always reduces *from* a value: this is because values of type `ST` can be seen as "frozen" computations, until they appear inside a `runST`. The expression e on the right hand-side of the rule above can be a reducible expression, which is reduced by using $K = \texttt{runST}\{[]\}$ as a context for the main reduction rule $\rightarrow$.

This operational semantics is new, therefore we include an example of how a simple program reduces. The program initializes a reference r to 3, then writes 7 into r and finally reads r.

$$\left\langle \emptyset, \texttt{runST}\ \left\{ \begin{array}{l} \texttt{bind}\,\texttt{ref}(3)\,\texttt{in}\,(\lambda\, r.\,\texttt{bind}\,(r \leftarrow 7)\,\texttt{in} \\ \\ (\lambda\, \_.\,\texttt{bind}\,!\,r\,\texttt{in}\,(\lambda\, x.\,\texttt{return}\,x))) \end{array} \right\} \right\rangle$$

The contents of the `runST` is a value, so we can use the rule above, and the context $\mathbb{K} = \texttt{bind}\,[]\,\texttt{in}\,\cdots$ to reduce $\langle \emptyset, \texttt{ref}(3)\rangle \rightsquigarrow_h \langle [l \mapsto 3], \texttt{return}\,l\rangle$ (for

some arbitrary l) and get:

$$\left\langle [l \mapsto 3], \texttt{runST} \begin{Bmatrix} \texttt{bind}\,(\texttt{return}\,l)\,\texttt{in}\,(\lambda\, r.\,\texttt{bind}\,(r \leftarrow 7)\,\texttt{in} \\ \\ (\lambda\, \_.\,\texttt{bind}\,!\,r\,\texttt{in}\,(\lambda\, x.\,\texttt{return}\,x))) \end{Bmatrix} \right\rangle$$

The contents of `runST` is still a value, and this time we use the empty context $\mathbb{K} = []$ and the rule for the `bind` of a `return`,

$$\langle [l \mapsto 3], \texttt{bind}\,(\texttt{return}\,l)\,\texttt{in}\,(\lambda\, r.\,\cdots)\rangle \;\rightsquigarrow_h\; \langle [l \mapsto 3], (\lambda\, r.\,\cdots)\; l\rangle$$

to get:

$$\langle [l \mapsto 3], \texttt{runST}\;\{(\lambda\, r.\,\texttt{bind}\,(r \leftarrow 7)\,\texttt{in}\,(\lambda\, \_.\,\texttt{bind}\,!\,r\,\texttt{in}\,(\lambda\, x.\,\texttt{return}\,x)))\; l\}\rangle$$

This time we use the context $K = \texttt{runST}\;\{[]\}$ and the rule for β-reduction to get:

$$\langle [l \mapsto 3], \texttt{runST}\;\{\texttt{bind}\,(l \leftarrow 7)\,\texttt{in}\,(\lambda\, \_.\,\texttt{bind}\,!\,l\,\texttt{in}\,(\lambda\, x.\,\texttt{return}\,x))\}\rangle$$

The situation is now the same as for the first two reduction steps and we reduce further to:

$$\langle [l \mapsto 7], \texttt{runST}\;\{\texttt{bind}\,(\texttt{return}\,())\,\texttt{in}\,(\lambda\, \_.\,\texttt{bind}\,!\,l\,\texttt{in}\,(\lambda\, x.\,\texttt{return}\,x))\}\rangle$$

and then, in two steps (rule for `bind` and `return`, then β-reduction):

$$\langle [l \mapsto 7], \texttt{runST}\;\{\texttt{bind}\,!\,l\,\texttt{in}\,(\lambda\, x.\,\texttt{return}\,x)\}\rangle$$

Finally we get:

$$\langle [l \mapsto 7], \texttt{runST}\;\{\texttt{return}\,7\}\rangle$$

and, from the rule for `runST` and `return` v:

$$\langle [l \mapsto 7], 7\rangle.$$

Having defined the operational semantics and the typing rules we can now define contextual refinement and equivalence. In this definition we write $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; 1)$ to express that C is a well-typed closing context (the remaining rules for this relation are completely standard).

Definition 6.2.1 (Contextual refinement and equivalence)**.** *We define contextual refinement* $\preceq_{\mathsf{ctx}}$ *and contextual equivalence* $\approx_{ctx}$ *as follows.*

$$\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau \;\triangleq\; \Xi \mid \Gamma \vdash e : \tau \;\wedge\; \Xi \mid \Gamma \vdash e' : \tau \;\wedge$$
$$\forall h, h', C.\; C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; 1) \;\wedge$$

$$(h, C[e])\downarrow \implies (h', C[e'])\downarrow$$

$$\Xi \mid \Gamma \vDash e \approx_{ctx} e' : \tau \triangleq \Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau \wedge \Xi \mid \Gamma \vDash e' \preceq_{\mathsf{ctx}} e : \tau.$$

where $(h, e)\downarrow \triangleq \exists h', v.\ (h, e) \to^* (h', v)$

6.3 Logical Relation

It is well-known that it is challenging to construct logical relations for languages with higher-order store because of the so-called type-world circularity (Ahmed, Appel, and Virga, 2002; Ahmed, 2004; Birkedal, Reus, Schwinghammer, Støvring, Thamsborg, and Yang, 2011). Other recent work has shown how this challenge can be addressed by using the original Iris logic to define logical relations for languages with higher-order store (Krebbers, Timany, and Birkedal, 2017; Krogh-Jespersen, Svendsen, and Birkedal, 2017). In fact, a key point is that Iris has enough logical features to give a direct inductive interpretation of the programming language types into Iris predicates.

The binary relations in Krebbers, Timany, and Birkedal (2017) and Krogh-Jespersen, Svendsen, and Birkedal (2017) were defined using Iris's built-in notion of Hoare triple and weakest precondition. This approach is, however, too abstract for our purposes: to prove the contextual refinements and equivalences for pure computations mentioned in the Introduction, we need to have more fine-grained control over how computations are related.

In this section we start by giving a gentle introduction to the base logic of Iris. Hereafter, we use the Iris base logic to define two new logical connectives called future modality and If-Convergent. We use these, instead of the weakest precondition used in Krebbers, Timany, and Birkedal (2017) and Krogh-Jespersen, Svendsen, and Birkedal (2017), when defining our binary logical relation.

We focus on properties that are necessary for understanding the key ideas of the definition of the logical relation; more technical details, including definitions and lemmas required for proving properties of the logical relation, are deferred to Appendix E.

6.3.1 An Iris Primer

Iris was originally presented as a framework for higher-order (concurrent) separation logic, with built-in notions of physical state (in our case heaps), ghost-

state (monoids) invariants and weakest preconditions, useful for Hoare-style reasoning about higher-order concurrent imperative programs (Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer, 2015). Subsequently, Iris was extended with a notion of higher-order ghost state (Jung, Krebbers, Birkedal, and Dreyer, 2016), i.e., the ability to store arbitrary higher-order separation-logic predicates in ghost variables. Recently, a simpler Iris *base logic* was defined, and it was shown how that base logic suffices for defining the earlier built-in concepts of invariants, weakest preconditions, and higher-order ghost state (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017).

In Iris one can quantify over the Iris types κ:

$$\kappa ::= 1 \mid \kappa \times \kappa \mid \kappa \to \kappa \mid \mathit{Expr} \mid \mathit{Val} \mid \mathbb{Z} \mid \mathbb{B} \mid \kappa \xrightarrow{\text{fin}} \kappa \mid$$

$$\text{finset}(\kappa) \mid \mathit{Monoid} \mid \mathit{Names} \mid \mathit{iProp} \mid \ldots$$

Here *Expr* and *Val* are the types of syntactic expressions and values of STLang, $\mathbb{Z}$ is the type of integers, $\mathbb{B}$ is the type of booleans, $\kappa \xrightarrow{\text{fin}} \kappa$ is the type of partial functions with finite support, $\text{finset}(\kappa)$ is the type of finite sets, *Monoid* is the type of monoids, *Names* is the type of ghost names, and *iProp* is the type of Iris propositions. A basic grammar for Iris propositions P is:

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\, \Phi \mid \exists x : \kappa.\, \Phi$$

$$\mid \triangleright P \mid \mu r.P \mid \Box P \mid \Rrightarrow P$$

The grammar includes the usual connectives of higher-order separation logic ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $*$, $\mathrel{-\!\!*}$, $\forall$ and $\exists$). In this grammar Φ is an Iris predicate, i.e., a term of type $\kappa \to \mathit{iProp}$ (for appropriate κ). The intuition is that the propositions denote sets of resources and, as usual in separation logic, $P * P'$ holds for those resources which can be split into two disjoint parts, with one satisfying P and the other satisfying P'. Likewise, the proposition $P \mathrel{-\!\!*} P'$ describes those resources which satisfy that, if we combine it with a disjoint resource satisfied by P we get a resource satisfied by P'. In addition to these standard connectives there are some other interesting connectives, which we now explain.

The $\triangleright$ is a modality, pronounced "later". It is used to guard recursively defined propositions: $\mu r.P$ is a well-defined guarded-recursive predicate only if r appears under a $\triangleright$ in P. The $\triangleright$ modality is an abstraction of step-indexing (Appel and McAllester, 2001; Appel, Melliès, Richards, and Vouillon, 2007; Dreyer, Ahmed, and Birkedal, 2011). In terms of step-indexing $\triangleright P$ holds if P holds a step later; hence the name. In Iris it can be used to define weakest preconditions and to guard impredicative invariants to avoid self-referential paradoxes (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). Here we simply use it to take a guarded fixed point when we give the interpretation of recursive types,

similarly to what was done in Dreyer, Ahmed, and Birkedal (2011). For any proposition P, we have that $P \vdash \triangleright P$. The later modality commutes with all of the connectives of higher-order separation logic, including quantifiers.

Another modality of the Iris logic is the "persistence" modality ($\Box$). This modality is used in Iris to capture a sublogic of knowledge (as opposed to resources) that obeys standard rules for intuitionistic higher-order logic. We say that P is *persistent* if $P \vdash \Box P$. Intuitively, $\Box P$ holds if P holds without asserting any exclusive ownership. Hence $\Box P$ is a duplicable assertion, i.e., we have $(\Box P) * (\Box P) \dashv\vdash \Box P$, where $\dashv\vdash$ is the logical equivalence of formulas. Hence persistent propositions are therefore duplicable. The persistence modality is idempotent, $\Box P \vdash \Box\Box P$, and also satisfies $\Box P \vdash P$. It (and by extension persistence) also commutes with all of the connectives of higher-order separation logic, including quantifiers.

The final modality we present in this section is the "update" modality[3] ($\Rrightarrow$). Intuitively, the proposition $\Rrightarrow P$ holds for resources that can be updated (through allocation, deallocation, or alteration) to resources that satisfy P, without violating the environment's knowledge or ownership of resources. We write $P \Rrightarrow Q$ as a shorthand for $P \mathrel{-\!\!*} \Rrightarrow Q$. The update modality is idempotent, $\Rrightarrow(\Rrightarrow P) \dashv\vdash \Rrightarrow P$.

6.3.2 Future Modality and If-Convergent

In this subsection we define two new constructs in Iris, which we will use to define the logical relation. The first construct, the future modality, will allow us to reason about what will happen in a "future world". The second construct, the If-Convergent predicate, will be used instead of weakest preconditions to reason about properties of computations.

Future Modality We define the *future modality* $\Rrightarrow\!\{\cdot\}\!\Rrightarrow$ as follows:

$$\Rrightarrow\!\{n\}\!\Rrightarrow P \triangleq (\Rrightarrow \triangleright)^n \Rrightarrow P$$

where $(\Rrightarrow \triangleright)^n$ is n times repetition of $\Rrightarrow \triangleright$. Intuitively, $\Rrightarrow\!\{n\}\!\Rrightarrow P$ expresses that n steps into the future, we can update our resources to satisfy P. We write $P \Rrightarrow\!\{n\}\!\Rrightarrow\! * Q$ as a shorthand for $P \mathrel{-\!\!*} \Rrightarrow\!\{n\}\!\Rrightarrow Q$.

[3] In (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017) this modality is called the *fancy* update modality. Technically, this modality comes equipped with certain "masks" but we do not discuss those here.

If-Convergent (IC) We define the *If-Convergent (IC)* predicate in Iris as follows:

$$\mathsf{IC}^{\gamma}\, e\, \{\!|v.\ Q|\!\} \triangleq \forall h_1, h_2, v, n.\ \langle h_1, e\rangle \to^{n} \langle h_2, v\rangle * \mathit{heap}_{\gamma}(h_1) \\ \gg\!\{n\}\!\models\!* \ \mathit{heap}_{\gamma}(h_2) * Q\ v$$

In general the number of steps, n, can also appear in Q but here we only present this slightly simpler version. The $\mathsf{IC}^{\gamma}\, e\, \{\!|v.\ Q|\!\}$ predicate expresses that, for any heap h_1, if (e, h_1) can reduce to (v, h_2) in n steps, and if we have ownership over h_1, then, n steps into the future, we will have ownership over the heap h_2, and the postcondition Q will hold.

A crucial feature of the IC predicate is that it allows us to use a ghost state name γ to keep track of the contents of the heap during the execution of e. This allows us to abstract away from the concrete heaps when reasoning about IC predicates[4]. Note that the IC predicate does *not* require that it is *safe* to execute the expression e: in particular, if e gets stuck (or diverges) in all heaps, then $\mathsf{IC}^{\gamma}\, e\, \{\!|v.\ Q|\!\}$ holds trivially.

The predicate $\mathit{heap}_{\gamma}(h_1)$ is a ghost state predicate stating ownership of a logical heap identified by the ghost state name γ (one can think of this as the usual ownership of a heap in separation logic). Ownership of a logical heap cell l is written as $\ell \mapsto_{\gamma} v$, and says that the heap identified by γ stores the value v at location ℓ. We show the precise definition of $\mathit{heap}_{\gamma_h}(h)$ and $\ell \mapsto_{\gamma} v$ in Appendix E; here we just highlight the properties that these abstract predicates enjoy:

$$\mathit{heap}_{\gamma}(h) * \ell \mapsto_{\gamma} v \Rightarrow h(l) = v \tag{6.1}$$

$$\mathit{heap}_{\gamma}(h) \wedge l \notin \mathrm{dom}(h) \ \Rrightarrow\!\!\!* \ \mathit{heap}_{\gamma}(h[l \mapsto v]) * \ell \mapsto_{\gamma} v \tag{6.2}$$

$$\mathit{heap}_{\gamma}(h) * \ell \mapsto_{\gamma} v \ \Rrightarrow\!\!\!* \ \mathit{heap}_{\gamma}(h[l \mapsto v']) * \ell \mapsto_{\gamma} v' \tag{6.3}$$

$$\ell \mapsto_{\gamma} v * \ell \mapsto_{\gamma} v' \ \Rightarrow \bot \tag{6.4}$$

Property (6.1) says that if we have ownership of a heap h and a location l pointing to v, both with the same ghost name γ, then we know that $h(l) = v$. Property (6.2) expresses that we can allocate a new location l in h, if l is not already in the domain of h. Finally, Property (6.3) says that we can update the value at location l, if we have both $\mathit{heap}_{\gamma}(h)$ and $\ell \mapsto_{\gamma} v$. Property (6.4) expresses exclusivity of the ownership of locations.

[4] This is related to the way the definition of weakest preconditions in Iris hides state (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017).

Akin to the way Hoare triples are defined in Iris using the weakest precondition, we define a new notion called IC triple as follows:

$$\{\!|\, P \,|\!\}\ e\ \{\!|\, v.\ Q \,|\!\}_\gamma \triangleq \square(P \mathrel{-\!\!*} \mathsf{IC}^\gamma\ e\ \{\!|\, v.\ Q \,|\!\})$$

The IC triple says, that given resources described by P, if e reduces in a heap identified by γ, then the post-condition Q will hold. Notice that the IC triple is a persistent predicate and is not allowed to own any exclusive resources.

6.3.3 Definition of the Logical Relation

We now have enough logical machinery to describe the logical relation (pedantically, it is a family of logical relations) shown in Figures 6.8 and 6.9. The logical relation is a binary relation, which allows us to relate pairs of expressions and pairs of values to each other. We will show that if two expressions are related in the logical relation, then the left hand side expression contextually approximates the right hand side expression. Therefore, we sometimes refer to the left hand side as the implementation and the right hand side as the specification.

The value relation $[\![\Xi \vdash \tau]\!]_\Delta$ is an Iris relation of type $(\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$ and, intuitively, it relates STLang values of type τ. The value relation is defined by induction on the type τ. Here, Ξ is an environment of type variables, and Δ is a semantic environment for these type variables, as is usual for languages with parametric polymorphism (Reynolds, 1983).

If τ is a ground type like $1, \mathbb{B}$ or $\mathbb{Z}$, two values are related at type τ if and only if they are equal (and compatible with the type). For instance, if τ is $\mathbb{Z}$, then $[\![\Xi \vdash \mathbb{Z}]\!]_\Delta(v, v') \triangleq v = v' \in \mathbb{Z}$.

For a product type of the form $\tau \times \tau'$, two values v and v' are related if and only if they both are pairs, and the corresponding components are related at their respective types:

$$[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v, v') \triangleq \exists w_1, w_2, w_1', w_2'.\ v = (w_1, w_2) \wedge v' = (w_1', w_2') \wedge$$
$$[\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') \wedge [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')$$

Note that the formula on the right hand side of $\triangleq$ is simply a formula in (the first order fragment of) Iris. The case of sum types is handled in a very similar fashion.

Two values v and v' are related at a function type $\tau \to \tau'$ if, given any two related values w and w' at type τ, the applications $v\ w$ and $v'\ w'$ are related at

Value relations:

$$[\![\Xi \vdash X]\!]_\Delta \triangleq (\Delta(X)).1$$

$$[\![\Xi \vdash 1]\!]_\Delta(v, v') \triangleq v = v' = ()$$

$$[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v, v') \triangleq v = v' \in \{\texttt{true}, \texttt{false}\}$$

$$[\![\Xi \vdash \mathbb{Z}]\!]_\Delta(v, v') \triangleq v = v' \in \mathbb{Z}$$

$$[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v, v') \triangleq \exists w_1, w_2, w_1', w_2'.\ v = (w_1, w_2) \wedge v' = (w_1', w_2') \wedge [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') \wedge [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')$$

$$[\![\Xi \vdash \tau + \tau']\!]_\Delta(v, v') \triangleq (\exists w, w'.\ v = \texttt{inj}_1\, w \wedge v' = \texttt{inj}_1\, w' \wedge [\![\Xi \vdash \tau]\!]_\Delta(w, w')) \vee (\exists w, w'.\ v = \texttt{inj}_2\, w \wedge v' = \texttt{inj}_2\, w' \wedge [\![\Xi \vdash \tau']\!]_\Delta(w, w'))$$

$$[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') \triangleq \Box\left(\forall (w, w').\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow \mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(v\ w, v'\ w')\right)$$

$$[\![\Xi \vdash \forall X.\,\tau]\!]_\Delta(v, v') \triangleq \Box\left(\forall f, r \in \mathcal{R}.\ \mathsf{persistent}(f) \Rightarrow \mathcal{E}\,[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f,r)}\,(v\ \_, v'\ \_)\right)$$

$$[\![\Xi \vdash \mu X.\,\tau]\!]_\Delta(v, v') \triangleq \mu f.\ \exists w, w'.\ v = \texttt{fold}\, w \wedge v' = \texttt{fold}\, w' \wedge \triangleright [\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f, \mathsf{toRgn}(\Delta, \mu X.\,\tau))}(w, w')$$

Figure 6.8: Binary logical relation

Value relations (continued):

$$[\![\Xi \vdash \texttt{STRef}\ \rho\ \tau]\!]_\Delta(v, v') \triangleq \exists \ell, \ell', r.\ v = \ell \wedge v' = \ell' \wedge \mathsf{isRgn}(\mathsf{toRgn}(\Delta, \rho), r) \\ * \mathsf{bij}(r, \ell, \ell')\ \ * \mathsf{rel}(r, \ell, \ell', [\![\Xi \vdash \tau]\!]_\Delta)$$

$$[\![\Xi \vdash \texttt{ST}\ \rho\ \tau]\!]_\Delta(v, v') \triangleq \forall \gamma_h, \gamma_h', h_1'.$$

$$\left\{ \left| \begin{array}{l} heap_{\gamma_h'}(h_1') * \mathsf{regions} * \\ \\ \mathsf{region}(\mathsf{toRgn}(\Delta, \rho), \gamma_h, \gamma_h') \end{array} \right| \right\}$$

$$\texttt{runST}\ \{v\}$$

$$\left\{ \left| \begin{array}{l} w.\ (h_1', \texttt{runST}\ \{v'\}) \Downarrow^{\gamma_h'}_{[\![\Xi \vdash \tau]\!]_\Delta(w, \cdot)} * \\ \\ \quad \mathsf{region}(\mathsf{toRgn}(\Delta, \rho), \gamma_h, \gamma_h') \end{array} \right| \right\}_{\gamma_h}$$

Expression relation:

$$\mathcal{E}\,\Phi(e, e') \triangleq \forall \gamma_h, \gamma_h', h_1'.\ \left\{ \left| heap_{\gamma_h'}(h_1') * \mathsf{regions} \right| \right\}$$

$$e$$

$$\left\{ \left| w.\ (h_1', e') \Downarrow^{\gamma_h'}_{\Phi(w, \cdot)} \right| \right\}_{\gamma_h}$$

Environment relation:

$$[\![\Xi \vdash \cdot]\!]^{\mathcal{G}}_\Delta(\vec{v}, \vec{v'}) \triangleq \top$$

$$[\![\Xi \vdash \Gamma, x : \tau]\!]^{\mathcal{G}}_\Delta(w\vec{v}, w'\vec{v'}) \triangleq [\![\Xi \vdash \tau]\!]_\Delta(w, w') * [\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v}, \vec{v'})$$

Logical relatedness:

$$\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v'}. \begin{array}{l} [\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v}, \vec{v'}) \Rightarrow \\ \mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\ \left(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}]\right) \end{array}$$

$$\mathsf{toRgn}(\Delta, \tau) \triangleq \begin{cases} \Delta(X).2 & \text{if } \tau = X \text{ is a type variable} \\ 1 & \text{otherwise} \end{cases}$$

Figure 6.9: Binary logical relation (continued)

type τ'. Notice that those latter two terms are expressions, not values; thus they have to be related under the expression relation $\mathcal{E}[\![\Xi \vdash \tau']\!]_\Delta$, which we will define later. Using Iris, the case for function types is defined as follows:

$$[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') \triangleq$$

$$\square\left(\forall(w, w').\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow \mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(v\ w, v'\ w')\right)$$

The $\square$ modality is used to ensure that $[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v')$ is *persistent* and hence duplicable. In fact, we will make sure that all predicates $[\![\Xi \vdash \tau]\!]_\Delta(v, v')$ are persistent. The intuition behind this is that the types of STLang just express duplicable knowledge (the type system is not a substructural type system involving resources).

Let us now discuss the case of polymorphic types. We use the semantic environment Δ, which maps type variables to pairs consisting of an Iris relation on values (the semantic value relation interpreting the type variable) and a region name (we use positive integers, $\mathbb{Z}^+$, to identify regions):

$$\Delta : \mathit{Tvar} \to (((\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}) \times \mathbb{Z}^+)$$

Thus, we simply define $[\![\Xi \vdash X]\!]_\Delta \triangleq \Delta(X).1$.

For type abstraction, two values v and v' are related at $\forall X.\,\tau$ when $v\ \_$ and $v'\ \_$ are related at τ, where the environments (Ξ and Δ) have been extended with X, and any persistent binary value relation f. (Recall that $v\ \_$ is the syntax for type application).

$$[\![\Xi \vdash \forall X.\,\tau]\!]_\Delta(v, v') \triangleq \square\left(\forall f.\ \mathsf{persistent}(f) \Rightarrow \mathcal{E}[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto f}(v\ \_, v'\ \_)\right)$$

The last case, before we get to the types associated to the ST monad, is the case of recursive types: two values are related at type $\mu X.\,\tau$ if they are of the form $\texttt{fold}\,w$ and $\texttt{fold}\,w'$ and, moreover, w and w' are related at τ, where the type variable X is added to the environments, and mapped in Δ to $([\![\Xi \vdash \mu X.\,\tau]\!]_\Delta, \mathsf{toRgn}(\Delta, \mu X.\,\tau))$ (ignore $\mathsf{toRgn}(\Delta, \mu X.\,\tau)$ for now):

$$[\![\Xi \vdash \mu X.\,\tau]\!]_\Delta(v, v') \triangleq \mu f.\ \Big(\exists w, w'.\ v = \texttt{fold}\,w \wedge v' = \texttt{fold}\,w' \wedge$$

$$\triangleright[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f, \mathsf{toRgn}(\Delta, \mu X.\,\tau))}(w, w')\Big)$$

Notice that we use a guarded recursive predicate in Iris, which is well-defined because the occurrence of f is guarded by the later modality $\triangleright$.

Before describing the cases for `STRef` $\rho\ \tau$ and `ST` $\rho\ \tau$ we touch upon the expression relation, which is defined independently of the value relation and has the following type:

$$\mathcal{E}\,\cdot : ((\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}) \to (\mathit{Expr} \times \mathit{Expr}) \to \mathit{iProp}$$

Intuitively, the expression relation $\mathcal{E}\,\Phi\,(e, e')$ holds for two expressions e and e' if e (the implementation) refines, or approximates, e' (the specification). That is, reduction steps taken by e can be simulated by zero or more steps in e'. We use IC triples to define the expression relation. The IC triples are unary and are used to express a property of the implementation expression e. We use the following Iris assertion in the postcondition of the IC triple to talk about the reductions in the specification expression e':

$$(h_1', e') \Downarrow_{\Phi}^{\gamma} \triangleq \exists h_2', v'.\ \langle h_1', e'\rangle \to_d^* \langle h_2', v'\rangle * \mathit{heap}_{\gamma}(h_2') * \Phi(v')$$

This assertion says that there exists a *deterministic* reduction from (h_1', e') to (h_2', v'), that the resulting heap h_2' is owned and the value satisfies Φ. The deterministic reduction relations, $\to_d$ and $\rightsquigarrow_d$, are defined by the same inference rules as $\to$ and $\rightsquigarrow$, except that the only non-deterministic rule, Alloc, is replaced by a deterministic one:

$$\textsc{det-Alloc}\ \frac{\ell = \mathit{min}(\mathit{Loc} \setminus \mathrm{dom}(h))}{\langle h, \mathtt{ref}(v)\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathtt{return}\,\ell\rangle}$$

The requirement that the reduction on the specification side is *deterministic* is used crucially in the proofs of the purity properties in §6.4. We emphasize that even with this requirement, we can still prove that logical relatedness implies contextual refinement (without requiring that STLang use deterministic reductions), essentially since we only require determinism on the specification side.

Thus, in more detail, the expression relation $\mathcal{E}\,\Phi\,(e, e')$ says that, when given full ownership of a heap h_1' for the specification side ($\mathit{heap}_{\gamma'}(h_1')$), if e reduces to a value w when given some heap h (quantified in IC), then a deterministic reduction on the specification side exists, and the resulting values are related. Notice that the heaps used for the implementation and specification side reductions are universally quantified, because we quantify over the ghost names γ_h, and γ_h', and that we do not require any explicit relationship between them. The persistent Iris assertion regions is responsible for keeping track of all allocated regions; it will be explained later.

For the value interpretation of `STRef` $\rho\ \tau$ and `ST` $\rho\ \tau$, the key idea is to tie each type ρ in an ST monad type (ρ in `ST` $\rho\ \tau$) to a semantic region name $r \in \mathbb{Z}^+$.

The association can be looked up using the function toRgn. Intuitively, a region r contains a collection of pairs of locations (one for the implementation side and one for the specification side) in one-to-one correspondence, together with a semantic predicate ϕ for each pair of locations in the region. The idea is that an implementation-side heap h and a specification-side heap h' satisfies a region r if, for any pair of locations (ℓ, ℓ') in r, we have values v and v', such that $h(\ell) = v$ and $h'(\ell') = v'$ and $\phi(v, v')$. All this information is contained in the predicate $\mathsf{region}(r, \gamma_h, \gamma'_h)$, where γ_h and γ'_h are the ghost names for the implementation and specification heap, respectively.

We have to maintain a one-to-one correspondence between locations because the operational semantics allows for comparison of locations. Given the one-to-one correspondence, we know that two locations on the implementation side are equal *if and only if* their two related counterparts on the specification side are.

We write $\mathsf{isRgn}(r, \rho)$ to say that r is the semantic region tied to ρ. We keep track of all regions by the $\mathsf{regions}$ assertion, which allows us to allocate new regions, as so:

$$\mathsf{regions} \;\; \text{==}\!\!\!* \;\; \exists r.\; \mathsf{region}(r, \gamma_h, \gamma'_h) \tag{6.5}$$

Notice that (6.5) gives back a fresh semantic region r. The $\mathsf{region}(r, \gamma_h, \gamma'_h)$ predicate allows for local reasoning about relatedness of two locations in a region r. We use a predicate $\mathsf{bij}(r, \ell, \ell')$, which in conjunction with region captures that ℓ and ℓ' are related by the one-to-one correspondence in r. Similarly, we use a predicate $\mathsf{rel}(r, \ell, \ell', \phi)$ in conjunction with region for local reasoning about the fact that values at locations ℓ and ℓ' in region r are related by predicate ϕ.

With this in mind, the definition of the value relation for $\texttt{STRef}\ \rho\ \tau$ is that there exists a semantic region r and locations ℓ and ℓ' in a bijection, $\mathsf{bij}(r, \ell, \ell')$, such that values pointed to by these locations are related by the relation corresponding to the type τ, asserted by $\mathsf{rel}(r, \ell, \ell', [\![\Xi \vdash \tau]\!]_\Delta)$.

Finally, (v, v') are related by $[\![\Xi \vdash \texttt{ST}\ \rho\ \tau]\!]_\Delta$ if, for any h_1 and h'_1 related in r ($\mathsf{region}(r, \gamma_h, \gamma'_h)$) along with some h_2 and w such that $\langle h_1, \texttt{runST}\ \{v\}\rangle \to^* \langle h_2, w\rangle$, then there is a heap h'_2 and a value w' such that we *afterwards* have $\langle h'_1, \texttt{runST}\ \{v'\}\rangle \to^*_d \langle h'_2, w'\rangle$ and $\mathsf{region}(r, \gamma_h, \gamma'_h)$ still holds. The intuitive meaning of the word *afterwards* refers to an application of the future modality (in the IC triple). Note that it is important that the semantic region r still holds after $\texttt{runST}\ \{v\}$ and $\texttt{runST}\ \{v'\}$ have been evaluated. This captures that encapsulated computations cannot modify the values of existing locations, but may allocate new locations (in new regions).

We have now completed the explanation of the value and expression relation for closed values and expressions. As usual for logical relations, we then relate

open terms by closing them by related substitutions, as specified according the environment relation $\mathcal{G}$, and finally relate them in the expression relation for closed terms, see the definition of $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e' : \tau$ in Figures 6.8 and 6.9.

6.3.4 Properties of the Logical Relation

To show the fundamental theorem and the soundness of the logical relation wrt. contextual approximation, we prove compatibility lemmas for all typing rules. Instead of working with the explicit definition of the IC triple, we make use of the following properties of IC:

Lemma 6.3.1 (Properties of IC)**.**

1. $\mathit{IC}^\gamma\ e\ \{\!|v.\ Q|\!\} * (\forall w.\ (Q\ w) \mathrel{-\!\!*} \mathit{IC}^\gamma\ K[w]\ \{\!|v.\ Q'\ v|\!\}) \vdash \mathit{IC}^\gamma\ K[e]\ \{\!|v.\ Q'|\!\}$
2. $\Rrightarrow (Q\ w) \vdash \mathit{IC}^\gamma\ w\ \{\!|v.\ Q|\!\}$
3. $(\forall v.\ (P\ v) \Rrightarrow\!\!\!\!\ast\ (Q\ v)) * \mathit{IC}^\gamma\ e\ \{\!|v.\ P|\!\} \vdash \mathit{IC}^\gamma\ e\ \{\!|v.\ Q|\!\}$
4. $\Rrightarrow \mathit{IC}^\gamma\ e\ \{\!|v.\ Q|\!\} \vdash \mathit{IC}^\gamma\ e\ \{\!|v.\ Q|\!\}$
5. $\mathit{IC}^\gamma\ e\ \{\!|v.\ \Rrightarrow Q|\!\} \vdash \mathit{IC}^\gamma\ e\ \{\!|v.\ Q|\!\}$
6. $(\forall h.\ \langle h, e\rangle \to \langle h, e'\rangle) * \triangleright\, \mathit{IC}^\gamma\ e'\ \{\!|v.\ Q|\!\} \vdash \mathit{IC}^\gamma\ e\ \{\!|v.\ Q|\!\}$
7. $\triangleright(\forall \ell.\ \ell \mapsto_\gamma v \Rrightarrow\!\!\!\!\ast\ Q\ \ell) \vdash \mathit{IC}^\gamma\ \mathtt{runST}\ \{\mathtt{ref}(v)\}\ \{\!|w.\ Q|\!\}$
8. $\triangleright\, \ell \mapsto_\gamma v * \triangleright(\ell \mapsto_\gamma v \Rrightarrow\!\!\!\!\ast\ Q\ v) \vdash \mathit{IC}^\gamma\ \mathtt{runST}\ \{!\,\ell\}\ \{\!|w.\ Q|\!\}$
9. $\triangleright\, \ell \mapsto_\gamma v' * \triangleright(\ell \mapsto_\gamma v \Rrightarrow\!\!\!\!\ast\ Q\ ()) \vdash \mathit{IC}^\gamma\ \mathtt{runST}\ \{\ell \leftarrow v\}\ \{\!|w.\ Q|\!\}$
10. $\mathit{IC}^\gamma\ \mathtt{runST}\ \{e\}\ \{\!|v.\ Q|\!\} * \Big(\forall w.\ (Q\ w) \mathrel{-\!\!*}$
    $\mathit{IC}^\gamma\ \mathtt{runST}\ \{\mathbb{K}[\mathtt{return}\, w]\}\ \{\!|v.\ Q'\ w|\!\}\Big) \vdash \mathit{IC}^\gamma\ \mathtt{runST}\ \{\mathbb{K}[e]\}\ \{\!|v.\ Q'|\!\}$

Items (1) and (2) above show that IC is a monad in the same way that weakest precondition is a monad, known as the Dijkstra monad. Item (3) allows one to strengthen the post-condition. Items (4) and (5) says that we can dispense with the update modality $\Rrightarrow$ for IC since the update modality is idempotent and IC is based on the update modality. Item (6) says that if a pure reduction from e to e' exists and later the postcondition Q will hold when reducing e', then Q will also hold when reducing e. Items (7),(8) and (9) are properties that allow to allocate, read and modify the heap, all expressing, that the post-condition

Q will hold, if the resources needed are given and Q holds for the updated resources. Finally, (10) captures the "bind" property for the RunST monad.

All the compatibility lemmas have been proved in the Coq formalization; here we just sketch the proof of the compatibility lemma for `runST`:

Lemma 6.3.2 (Compatibility for runST)**.** *Suppose* $\Xi, X \mid \Gamma \vDash e \preceq_{\mathsf{log}} e' : \mathtt{ST}\ X\ \tau$ *and* $\Xi \vdash \tau$*. Then*

$$\Xi \mid \Gamma \vDash \mathtt{runST}\ \{e\} \preceq_{\mathsf{log}} \mathtt{runST}\ \{e'\} : \tau$$

Proof Sketch. We prove that for any f and r that

$$[\![\Xi, X \vdash \mathtt{ST}\ X\ \tau]\!]_{\Delta, X \mapsto (f,r)}(v, v') \ \text{ implies } \ \mathcal{E}\,[\![\Xi \vdash \tau]\!]_{\Delta}\,(\mathtt{runST}\ \{v\}\,, \mathtt{runST}\ \{v'\})$$

The lemma follows from the assumption that e and e' are suitably related. Assume we have regions, ghost names for the implementation and specification side, γ_h and γ'_h, and $heap_{\gamma'_h}(h'_1)$ for some h'_1. We are to show:

$$\mathsf{IC}^{\gamma_h}\ \mathtt{runST}\ \{v\} \left\{ \left| w.\ \exists h'_2, w'.\ \langle h'_1, \mathtt{runST}\ \{v'\}\rangle \to_d^* \langle h'_2, w'\rangle * \right. \right. \\ \left. \left. heap_{\gamma'_h}(h'_2) * [\![\Xi \vdash \tau]\!]_{\Delta}(w, w') \right| \right\}$$

Using (6.5) with regions we know there exists a fresh semantic region r and that the predicate $\mathsf{region}(r, \gamma_h, \gamma'_h)$ holds for r. We then instantiate our assumption by the unit relation $[\![\Xi \vdash 1]\!]_{\Delta}$ and r to get $[\![\Xi, X \vdash \mathtt{ST}\ X\ \tau]\!]_{\Delta, X \mapsto ([\![\Xi \vdash 1]\!]_{\Delta}, r)}(v, v')$.

By the definition of the value relation for the type $\mathtt{ST}\ X\ \tau$, we get that if we give a starting specification heap $heap_{\gamma'_h}(h'_1)$ and $\mathsf{region}(r, \gamma_h, \gamma'_h)$, then we have `runST` $\{v\}$ reduces to a value w, and there exist a reduction on the specification side producing w' such that w and w' are related by $[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto ([\![\Xi \vdash 1]\!], \rho)}$. Moreover, we also get the ownership of the resulting specification heap $heap_{\gamma_h}(h'_2)$.

By Lemma 6.3.1 (3), it suffices to show: $\Rightarrow\!\exists h'_2, w'.\ \langle h'_1, \mathtt{runST}\ \{v'\}\rangle \to_d^* \langle h'_2, w'\rangle * heap_{\gamma'_h}(h'_2) * [\![\Xi \vdash \tau]\!]_{\Delta}(w, w')$. The only thing that we do not immediately have from our assumption is $[\![\Xi \vdash \tau]\!]_{\Delta}(w, w')$, we only have that w and w' are related in a larger environment. However since X does not appear free in τ (which follows from $\Xi \vdash \tau$) it follows by induction on τ that $[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto ([\![\Xi \vdash 1]\!], r)}(w, w') \dashv\vdash [\![\Xi \vdash \tau]\!]_{\Delta}(w, w')$ which concludes the proof. □

Notice that in the above proof we start out with two completely unrelated heaps for the specification and the implementation side since these are universally

quantified inside the IC triple. We then establish a *trivial* relation between them by creating a new *empty* region. We extend and maintain this relation during the simulation of the stateful expressions on both sides. This is in essence the reason why our expression relations need not assume (or guarantee at the end) any relation between the heaps on the implementation and specification sides.

Using the compatibility lemmas, we can prove the following two theorems.

Theorem 6.3.3 (Fundamental theorem)**.** $\Xi \mid \Gamma \vdash e : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : \tau$

Theorem 6.3.4 (Soundness of logical relation)**.**

$$\Xi \mid \Gamma \vdash e : \tau \wedge \Xi \mid \Gamma \vdash e' : \tau \wedge \Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e' : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau$$

6.4 Proving Contextual Refinements and Equivalences

In this section we show how to prove the contextual refinements and equivalences mentioned in the Introduction. For the sake of illustration we present the proofs of Neutrality and one side of the Commutativity theorems in moderate detail — the proofs of these two cases demonstrate the key techniques that are also used to show the remaining contextual refinements and equivalences from the Introduction. For the remaining theorems, we only sketch their proofs at a higher level of abstraction. Readers who are eager to see all proofs in all their details are thus referred to our Coq formalization. Details pertaining to the Coq formalization can be found in Appendix E.

Theorem 6.4.1 (Neutrality)**.** *If* $\Xi \mid \Gamma \vdash e : 1$ *then* $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} () : 1$

Proof Sketch. By the fundamental theorem we have $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : 1$. We show that this implies $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} () : 1$. The final result follows from the soundness theorem.

By unfolding the IC predicate, we get the assumption that $\langle h_1, e\rangle \to^* \langle h_2, v\rangle$, including the ownership of $heap_{\gamma_h}(h_1)$ and $heap_{\gamma'_h}(h'_1)$, and have to prove that[5] $\langle h'_1, ()\rangle$ reduces deterministically to a value w (and some heap) and that (v, w) are in the value relation for the unit type. We proceed by allocating a copy of h'_1, obtaining $heap_\gamma(h'_1)$ for some fresh γ. We use this together with our assumptions, notably $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : 1$, to get that $\langle h'_1, e\rangle \to_d^* \langle h'_2, v'\rangle$ for some v' and h'_2 such that (v, v') are related in the value relation for the unit type, i.e., $v = v' = ()$, $heap_{\gamma_h}(h_2)$ and $heap_\gamma(h'_2)$. Notice that we have, crucially, retained

[5] We ignore the future modality for the sake of simplicity.

the ownership of $heap_{\gamma'_h}(h'_1)$ and have only updated the freshly allocated copy of h'_1 with the fresh name γ. We are allowed to do this because the relatedness of expressions, as in $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : 1$, universally quantifies over ghost names for the specification and implementation side heaps. We conclude the proof by noting that since () is a value, we have, trivially, $\langle h'_1, () \rangle \to_d^* \langle h'_1, () \rangle$ and that $(v, ())$ are related at the unit type. □

Theorem 6.4.2 (Commutativity)**.** *If* $\Xi \mid \Gamma \vdash e_1 : \tau_1$ *and* $\Xi \mid \Gamma \vdash e_2 : \tau_2$ *then*

$$\Xi \mid \Gamma \vDash \mathtt{let}\, x = e_2\, \mathtt{in}\, (e_1, x) \;\approx_{ctx} (e_1, e_2) \;:\; \tau_1 \times \tau_2$$

Proof Sketch. We only show $\Xi \mid \Gamma \vDash \mathtt{let}\, x = e_2\, \mathtt{in}\, (e_1, x) \;\preceq_{\mathsf{log}} (e_1, e_2) \;:\; \tau_1 \times \tau_2$, the other direction is similar. Unfolding the IC predicate we get the assumption that $\langle h_1, \mathtt{let}\, x = e_2\, \mathtt{in}\, (e_1, x) \rangle \to^* \langle h_2, v \rangle$ for some h_2 and v, the ownership of $heap_{\gamma_h}(h_1)$ and $heap_{\gamma'_h}(h'_1)$ and we have to prove that $\langle h'_1, (e_1, e_2) \rangle \to_d^* \langle h'_2, v' \rangle$ for some h'_2 and v', and that (v, v') are in the value relation for $\tau \times \tau'$. From the first assumption, we can conclude that $\langle h_1, e_2 \rangle \to^* \langle h_3, v_2 \rangle$, $\langle h_3, e_1 \rangle \to^* \langle h_2, v_1 \rangle$ and that $v = (v_1, v_2)$.

We proceed by allocating a fresh copy of h_3 (the heap in the middle of execution of the implementation side) with the fresh name γ, $heap_\gamma(h_3)$ and also a fresh $heap_{\gamma'}(h'_1)$ (the heap at the beginning of execution of the specification side). Notice that these are heaps (on either side) immediately before executing e_1. We use these freshly allocated heaps together with $\Xi \mid \Gamma \vDash e_1 \preceq_{\mathsf{log}} e_1 : \tau_1$ (which follows from the fundamental theorem) to conclude[6] $\langle h'_1, e_1 \rangle \to_d^* \langle h'_3, v'_1 \rangle$ for some v'_1 and h'_3.

Now we have the information about the starting heap for execution of e_2 on the specification side. Thus, we are ready to simulate the execution of e_2 on both sides. Note that the order of simulations is dictated by the order on the implementation side as we have to prove that the implementation side is simulated by the specification side.

To simulate e_2 we proceed by allocating a fresh copy of h'_3 (the heap immediately before executing e_2 on the specification side) with a fresh name γ'', $heap_{\gamma''}(h'_3)$. We use this, together with $heap_{\gamma_h}(h_1)$ (which we originally got by unfolding the IC predicate) and $\Xi \mid \Gamma \vDash e_2 \preceq_{\mathsf{log}} e_2 : \tau_2$ (which we know from the fundamental theorem). We can do this as we know $\langle h_1, e_2 \rangle \to^* \langle h_3, v_2 \rangle$. This allows us to conclude that $\langle h'_3, e_2 \rangle \to_d^* \langle h'_2, v'_2 \rangle$ for some h'_2 and v'_2, the ownership of $heap_{\gamma''}(h'_2)$ and $heap_{\gamma_h}(h_3)$ together with the fact that (v_2, v'_2) are related at type τ_2.

[6] For simplicity, we are ignoring some manipulations involving the future modality.

Now we are ready to simulate e_1 on both sides. We use $\Xi \mid \Gamma \vDash e_1 \preceq_{\mathsf{log}} e_1 : \tau_1$ (which we know from the fundamental theorem) together with $heap_{\gamma_h}(h_3)$ (from simulating e_2) and $heap_{\gamma'_h}(h'_1)$ (which we had as an assumption from the definition relatedness). We can do this because we know that $\langle h_3, e_1\rangle \to^* \langle h_2, v_1\rangle$. This allows us to conclude that $\langle h'_1, e_1\rangle \to^*_d \langle h''_3, v''_1\rangle$ for some h''_3 and v''_1, the ownership of $heap_{\gamma'_h}(h''_3)$ and $heap_{\gamma_h}(h_2)$ together with the fact that (v_1, v''_1) are related at type τ_1. It follows from the determinism of reduction on the specification side that $h'_3 = h''_3$ and $v'_1 = v''_1$.

The only thing we need to conclude the proof is the ownership of $heap_{\gamma'_h}(h_2)$ (the heap at the end of execution of the specification side) whereas we own $heap_{\gamma'_h}(h''_3)$ which is the heap of the specification side after execution of e_1 and before execution of e_2. However, using some resource reasoning (which depends on details explained in Appendix E), we can conclude that $h''_3 \subseteq h_2$. This in turn allows us to update our heap resource to get $heap_{\gamma'_h}(h_2)$, which concludes the proof. □

The proof sketches of the two theorems above show that the true expressiveness of our logical relation comes from the fact that the expression relation quantifies over the names of resources used for the heaps on the specification and implementation sides. This allows us to allocate fresh instances of ghost resources corresponding to the heaps (for any of the two sides) and simulate the desired part of the program. This is the reason why we can prove such strong equations as Commutativity, Idempotency, Hoisting, etc. The proof of Commutativity above also elucidates the use of deterministic reduction for the specification side.

Theorem 6.4.3 (Idempotency)**.** *If* $\Xi \mid \Gamma \vdash e : \tau$ *then* $\Xi \mid \Gamma \vDash \mathtt{let}\, x = e\, \mathtt{in}\, (x, x) \approx_{ctx} (e, e) : \tau \times \tau$

Proof Sketch. We show the contextual equivalence, by proving logical relatedness in both directions. For the left-to-right direction, we allocate a fresh heap and simply simulate twice on the specification side using the same reduction on the implementation side. For the other direction, we simulate the same reduction on the specification side twice for the two different reductions on the implementation side. For the latter we conclude, by determinism of reduction on the specification side, that the two reductions coincide. □

Theorem 6.4.4 (Rec Hoisting)**.** *If* $\Xi \mid \Gamma \vdash e_1 : \tau$ *and* $\Xi \mid \Gamma, y : \tau, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e_2 : \tau_2$ *then*

$$\Xi \mid \Gamma \vDash \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{rec}\, f(x) = e_2 \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = \mathtt{let}\, y = e_1\, \mathtt{in}\, e_2 : \tau_1 \to \tau_2$$

Proof Sketch. The proof of this theorem is quite tricky, in particular because the the number of operational steps do not match up for the function bodies on the implementation and specification sides. We do not delve into those issues here, but concentrate instead on the high-level structure of the proof.

We prove three different contextual refinements, such that their composition gives us the desired contextual refinement in the theorem. These three contextual refinements are:

(a) $\texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = e_2 \preceq_{\mathsf{ctx}} \texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, z = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$

(b) $\texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, z = e_1\, \texttt{in}\, e_2 \preceq_{\mathsf{ctx}} \texttt{let}\, z = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$

(c) $\texttt{let}\, z = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 \preceq_{\mathsf{ctx}} \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$ where z is a fresh variable.

We prove (a) by proving the corresponding logical relatedness. Since e_1 reduces to a value we know that it will reduce deterministically to some value under any heap on the specification side. We prove (c) also by the corresponding logical relatedness which is rather trivial to prove.

To prove (b) we show the corresponding logical relatedness for a slightly stronger logical relation; $\preceq^{NN}_{\mathsf{log}}$. The NN-logical relation is defined entirely similarly to the primary logical relation above except that the specification side is required to deterministically reduce to a value *in the same number of steps* as the implementation side. Notice that the proofs of the fundamental theorem and soundness for NN-logical relation are very similar to those of the primary logical relation.

Formally, for (b) we show

$$\texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, z = e_1\, \texttt{in}\, e_2$$

$$\preceq^{NN}_{\mathsf{log}} \quad \texttt{let}\, z = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 : \tau' \to \tau''$$

This logical relatedness is in fact rather easy to show if we know that all reductions of e_1 (on either side) take the same number of steps. This is precisely why we use the NN-logical relation: By the fundamental theorem of the NN-logical relation we know that $e_1 \preceq^{NN}_{\mathsf{log}} e_1 : \tau' \to \tau''$ and hence we can conclude that both outer reductions (on either side) take the same number of steps, say n. Similarly we know that both reductions of e_1 inside the functions also take the same number of steps, say m. Hence, by allocating appropriate heaps, we can show that the outer reduction of e_1 on the implementation side takes the

same number steps as that of the reduction of the inner one on the specification side. This shows, by determinism of reduction on the specification side, that $n = m$, which allows us to conclude the proof. □

Theorem 6.4.5 (η expansion for Rec)**.** *If* $\Xi \mid \Gamma \vdash e : \tau_1 \to \tau_2$ *then* $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = e\ x : \tau_1 \to \tau_2$

Proof Sketch. We prove this theorem by proving the following three contextual refinements.

(a) $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} \mathtt{let}\, y = e\, \mathtt{in}\, \mathtt{rec}\, f(x) = (y\ x) : \tau \to \tau'$

(b) $\Xi \mid \Gamma \vDash \mathtt{let}\, y = e\, \mathtt{in}\, \mathtt{rec}\, f(x) = (y\ x) \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = \mathtt{let}\, y = e\, \mathtt{in}\, (y\ x) : \tau \to \tau'$

(c) $\Xi \mid \Gamma \vDash \mathtt{rec}\, f(x) = \mathtt{let}\, y = e\, \mathtt{in}\, (y\ x) \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = (e\ x) : \tau \to \tau'$

Refinements (a) and (c) follow rather easily from their corresponding logical relatedness while case (b) is an instance of rec Hoisting above. For (c) notice that f does not appear free in e. □

Theorem 6.4.6 (β reduction for λ)**.** *If* $\Xi \mid \Gamma, x : \tau_1 \vdash e_1 : \tau_2$ *and* $\Xi \mid \Gamma \vdash e_2 : \tau_1$ *then*

$$(\lambda\, x.\, e_1)\ e_2 \ \preceq_{\mathsf{ctx}}\ e_1[e_2/x] \ :\ \tau$$

Proof Sketch. By induction on the typing derivation of e_1; for each case we use appropriate contextual refinements proven by (using the induction hypothesis if necessary) some of the contextual refinement theorems stated above and some instances of logical relatedness. We only present a couple cases here.

Case $e_1 = \mathtt{inj}_i\, e$ The induction hypothesis tells us that $\Xi \mid \Gamma \vDash (\lambda\, x.\, e)\ e_2 \preceq_{\mathsf{ctx}} e[e_2/x] : \tau_i$ and we have to show that $\Xi \mid \Gamma \vDash (\lambda\, x.\, \mathtt{inj}_i\, e)\ e_2 \preceq_{\mathsf{ctx}} (\mathtt{inj}_i\, e)[e_2/x] : \tau_1 + \tau_2$. Notice that it is easy to prove (using the fundamental theorem) that $\Xi \mid \Gamma \vDash (\lambda\, x.\, \mathtt{inj}_i\, e)\ e_2 \preceq_{\mathsf{log}} \mathtt{inj}_i\, ((\lambda\, x.\, e)\ e_2) : \tau_1 + \tau_2$ The final result follows by the induction hypothesis, transitivity of contextual refinement and the fact that contextual refinement is a congruence relation.

Case $e_1 = \mathtt{rec}\, f(y) = e$ The induction hypothesis tells us that $\Xi \mid \Gamma, y : \tau_1, f : \tau_1 \to \tau_2 \vDash (\lambda\, x.\, e)\ e_2 \preceq_{\mathsf{ctx}} e[e_2/x] : \tau_2$ and we have to show that $\Xi \mid \Gamma \vDash (\lambda\, x.\, (\mathtt{rec}\, f(y) = e))\ e_2 \preceq_{\mathsf{ctx}} (\mathtt{rec}\, f(y) = e)[e_2/x] : \tau_1 \to \tau_2$ or equivalently (by simply massaging the terms) $\Xi \mid \Gamma \vDash \mathtt{let}\, x = e_2\, \mathtt{in}\, (\mathtt{rec}\, f(y) = e) \preceq_{\mathsf{ctx}}$

$(\mathtt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$. By rec Hoisting and transitivity of contextual refinement, it suffices to show $\Xi \mid \Gamma \vDash (\mathtt{rec}\, f(y) = \mathtt{let}\, x = e_2\, \mathtt{in}\, e) \preceq_{\mathsf{ctx}} (\mathtt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$ which easily follows from the induction hypothesis and the fact that contextual refinement is a congruence relation. □

We omit the theorems of hoisting and η-expansion for polymorphic terms as they are fairly similar in statement and proof to their counterparts for recursive functions. We also omit β-reduction for polymorphic terms and recursive functions. The former follows directly from the corresponding logical relatedness and the latter follows from β-reduction for λ's and rec-unfolding: if $\Xi \mid \Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2$, then

$$\Xi \mid \Gamma \vDash \mathtt{rec}\, f(x) = e \;\preceq_{\mathsf{ctx}}\; \lambda\, x.\, e'[(\mathtt{rec}\, f(x) = e')/f] : \tau_1 \to \tau_2,$$

which is a consequence of the corresponding logical relatedness.

Theorem 6.4.7 (Equations for stateful computations)**.** *See Figure 6.3.*

Proof. Left identity follows by proving both logical relatednesses. Right identity is proven as follows using equational reasoning:

$$\begin{aligned}
e_2\ e_1 &\preceq_{\mathsf{ctx}} \mathtt{let}\, x = e_2\, \mathtt{in}\, \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, (\mathtt{return}\, y)\, \mathtt{in} \\
&\quad \mathtt{let}\, z = x\ y\, \mathtt{in}\, (\lambda _.\, z) \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, x = e_2\, \mathtt{in}\, \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, (\mathtt{return}\, y)\, \mathtt{in} \\
&\quad (\lambda _.\, \mathtt{let}\, z = x\ y\, \mathtt{in}\, z) \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, x = e_2\, \mathtt{in}\, \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, (\mathtt{return}\, y)\, \mathtt{in}\, x \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{let}\, x = e_2\, \mathtt{in}\, \mathtt{bind}\, (\mathtt{return}\, y)\, \mathtt{in}\, x \\
&\preceq_{\mathsf{ctx}} \mathtt{bind}\, (\mathtt{return}\, e_1)\, \mathtt{in}\, e_2 : \mathtt{ST}\ \rho\ \tau
\end{aligned}$$

Here the second equation is by rec Hoisting and the fourth by a variant of commutativity. The rest follow by proving the corresponding logical relatedness. Associativity is proven as follows using equational reasoning:

$$\begin{aligned}
&\mathtt{bind}\, (\mathtt{bind}\, e_1\, \mathtt{in}\, e_2)\, \mathtt{in}\, e_3 \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, y\, \mathtt{in}\, \mathtt{let}\, z = (e_2, e_3)\, \mathtt{in}\, (\lambda\, x.\, \mathtt{bind}\, (\pi_1\, z)\ x\, \mathtt{in}\, \pi_2\, z) \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, y\, \mathtt{in}\, (\lambda\, x.\, \mathtt{let}\, z = (e_2, e_3)\, \mathtt{in}\, \mathtt{bind}\, (\pi_1\, z)\ x\, \mathtt{in}\, \pi_2\, z)
\end{aligned}$$

$$\preceq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, y\, \mathtt{in}\, (\lambda x.\, \mathtt{let}\, z_1 = e_2\, \mathtt{in}$$

$$\mathtt{let}\, z_2 = e_3\, \mathtt{in}\, \mathtt{let}\, z_3 = (z_1\ x)\, \mathtt{in}\, \mathtt{bind}\, z_3\, \mathtt{in}\, z_2)$$

$$\preceq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{bind}\, y\, \mathtt{in}\, (\lambda x.\, \mathtt{let}\, z_1 = e_2\, \mathtt{in}$$

$$\mathtt{let}\, z_3 = (z_1\ x)\, \mathtt{in}\, \mathtt{let}\, z_2 = e_3\, \mathtt{in}\, \mathtt{bind}\, z_3\, \mathtt{in}\, z_2)$$

$$\preceq_{\mathsf{ctx}} \mathtt{bind}\, e_1\, \mathtt{in}\, (\lambda x.\, \mathtt{bind}\, (e_2\ x)\, \mathtt{in}\, e_3) : \mathtt{ST}\ \rho\ \tau$$

Here the second equation is by rec Hoisting and the fourth by a variant of commutativity. The rest follow by proving the corresponding logical relatedness. □

6.5 Related work

The most closely related work is the original seminal work of Launchbury and Peyton Jones (1994), which we discussed and related to in the Introduction. In this section we discuss other related work.

Moggi and Sabry (2001) showed type soundness of calculi with `runST`-like constructs, both for a call-by-value language (as we consider here) and for a lazy language. The type soundness results were shown with respect to operational semantics in which memory is divided into regions: a `runST`-encapsulated computation always start out in an empty heap and the final heap of such a computation is thrown away. Thus their type soundness result does capture some aspects of encapsulation. However, the models in *loc. cit.* are not relational and therefore not suitable for proving relational statements such as our theorems above. The authors write: "*Indeed substantially more work is needed to establish soundness of equational reasoning with respect to our dynamic semantics (even for something as unsurprising as β-equivalence)*" (Moggi and Sabry, 2001).

In contrast to Moggi and Sabry (2001), who also considered type soundness for a call-by-need language, we only develop our model for a call-by-value language. For call-by-need one would need to keep track of the dependencies between effectful operations in the operational semantics and *only* evaluate them if they contribute to the end result. These dependencies would also have to be reflected in the logical relations model. It is not clear how difficult that would be and we believe it deserves further investigation.

It was pointed out already in Launchbury and Peyton Jones (1994) that there seems to be a connection between encapsulation using `runST` and effect masking in type-and-effect systems à la Gifford and Lucassen (1986). This connection

was formalized by Semmelroth and Sabry (1999), who showed how a language with a simplified type-and-effect system with effect masking can be translated into a language with `runST`. Moreover, they showed type soundness on their language with `runST` with respect to an operational semantics. In contrast to our work, they did not investigate relational properties such as contextual refinement or equivalence.

Benton *et al.* have investigated contextual refinement and equivalence for type-and-effect systems in a series of papers (Benton and Buchlovsky, 2007; Benton, Kennedy, Beringer, and Hofmann, 2007, 2009; Benton, Kennedy, Hofmann, and Beringer, 2006) and their work was extended by Thamsborg and Birkedal (2011) to a language with higher-order store, dynamic allocation and effect masking. These papers considered soundness of some of the contextual refinements and equivalences for pure computations that we have also considered in this paper, but, of course, with very different assumptions, since the type systems in *loc. cit.* were type-and-effect systems. Thus, as an alternative to the approach taken in this paper, one could also imagine trying to prove contextual equivalences in the presence of `runST` by translating the type system into the language with type-and-effects used in Thamsborg and Birkedal (2011) and then appeal to the equivalences proved there. We doubt, however, that such an alternative approach would be easier or better in any way. The logical relation that we define in this paper uses an abstraction of regions and relates regions to the concrete global heap used in the operational semantics. At a very high level, this is similar to the way regions are used as an abstraction in the models for type-and-effect systems, e.g., in Thamsborg and Birkedal (2011). However, since the models are for different type systems, they are, of course, very different in detail. One notable advance of the current work over the models for type-and-effect systems, e.g., the concrete step-indexed model used in Thamsborg and Birkedal (2011), is that our use of Iris allows us to give more abstract proofs of the fundamental lemma for contextual refinements than a more low-level concrete step-indexed model would.

Recently Iris has been used in other works to define logical relations for different type systems than the one we consider here (Krebbers, Timany, and Birkedal, 2017; Krogh-Jespersen, Svendsen, and Birkedal, 2017). The definitions of logical relations in those works have used Iris's weakest preconditions $\mathsf{wp}\, e\, \{v.\ P\}$ to reason about computations. Here, instead, we use our if-convergence predicate, $\mathsf{IC}^{\gamma}\, e\, \{\!|v.\ P|\!\}$. One of the key technical differences between the weakest precondition predicate and the if convergence predicate is that the latter keeps explicit track of the ghost variable γ used for heap. This allows us to reason about different (hypothetical) runs of the same expression, a property we exploit in the proofs of contextual refinements in §6.4.

6.6 Conclusion and Future Work

We have presented a logical relations model of STLang, a higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style *ST* monad type with `runST`. To the best of our knowledge, this is the first model which can be used to show that `runST` provides proper encapsulation of state, in the sense that a number of contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of stateful computations encapsulated using `runST`. We defined our logical relation in Iris, a state-of-the-art program logic. This greatly simplified the construction of the logical relation, e.g., because we could use Iris's features to deal with the well-known type-world circularity. Moreover, it provided us with a powerful logic to reason in the model. Our logical relation and our proofs of contextual refinements used several new technical ideas: in the logical relation, e.g., the linking of the region abstraction to concrete heaps and the use of determinacy of evaluation on the specification side; and, in the proof of contextual refinements, e.g., the use of a helper-logical relation for reasoning about equivalence of programs using the same number of steps on the implementation side and the specification side. Finally, we have used and extended the Iris implementation in Coq to formalize our technical development and proofs in Coq.

Future work Future work includes developing a model for a call-by-need variant of STLang. In the original paper (Launchbury and Peyton Jones, 1994), Launchbury and Peyton Jones argue that it would be useful to have a combinator for parallel composition of stateful programs, as opposed to the sequential composition provided by the monadic bind combinator. One possible direction for future work is to investigate the addition of concurrency primitives in the presence of encapsulation of state. It is not immediately clear what the necessary adaptations are for keeping the functional language pure. It would be interesting to investigate whether a variation of the parallelization theorem studied for type-and-effect systems in Krogh-Jespersen, Svendsen, and Birkedal (2017) would hold for such a language.

Acknowledge The last author would like to thank Georg Neis for early discussions about the topic of this paper. We would like to thank the anonymous referees for their invaluable constructive comments. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU), by the EUTypes funding ECOST-STSM-CA15123-150816-080859 and by the Flemish Research Fund grants G.0058.13 (until June 2017) and G.0962.17N (since July 2017).

Chapter 7

Logical Relations for Relational Verification of Concurrent Programs with Continuations

Concurrent higher-order imperative programming languages with continuations are very flexible and allow for the implementation of sophisticated programming patterns. For instance, researchers and practitioners have argued that the implementation of web servers can be simplified by using a programming pattern based on continuations. This programming pattern can, in particular, help simplify keeping track of the state of clients interacting with the server. However, such advanced programming programming languages are very challenging to reason about.

In this paper we present the first completely formalized tool for interactive mechanized relational verification of programs written in a concurrent higher-order imperative programming language with continuations (`call/cc` and `throw`). In more detail, we develop a novel logical relation which can be used to give mechanized proofs of contextual refinement. We use our method on challenging examples and prove, *e.g.*, that a rudimentary web server implemented using the continuation-based pattern is contextually equivalent to one implemented without the continuation-based pattern.

7.1 Introduction

It is well-known that web servers are intricate to program, in particular because one has to keep track of the complex evolution of the state of clients. Clients can refresh pages, press back and forward buttons of the browser, and so forth. Both researchers (Flatt, 2017; Krishnamurthi, Hopkins, McCarthy, Graunke, Pettyjohn, and Felleisen, 2007; Queinnec, 2004) and practitioners (*Written in Racket*; Might, 2017) have therefore advocated that one can simplify web server implementations considerably by using explicitly captured (using `call/cc`) server-side continuations. The point is that using continuations simplifies the book-keeping of the clients' state and hence allows for a more direct style implementation of web servers, where the interaction with clients can be programmed as though one was communicating through a console.

This *continuation-based* approach to web server implementation is in contrast to the perhaps more common practice, which we refer to as *state-storing*, where for every request, the server needs to analyze its internally stored state along with the client request in order to determine the proper response. In the *continuation-based* approach, the server simply resumes its internally stored continuation when it gets a new request from the client. The Racket web development community (Flatt, 2017) is probably the most prominent user of continuation-based servers.

Continuation-based servers make use of sophisticated programming language features, in particular continuations and concurrency, each of which are known to be very difficult to model and reason about. In this paper, we develop a new model for reasoning about concurrent higher-order imperative programs with continuations. Specifically, we develop a new logical relations model for proving contextual equivalence of programs written in $\mathsf{F}^{\mu,ref}_{conc,cc}$, a call-by-value programming language featuring concurrency, impredicative polymorphism, recursive types, dynamically allocated higher-order store and first-class continuations with `call/cc` and `throw` primitives. We employ this logical relations model to prove (contextual) equivalence of two simple web server implementations: a continuation-based one and a state-storing one.

We define our logical relations model in a variant of the Iris program logic framework (Jung, Krebbers, Birkedal, and Dreyer, 2016; Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer, 2015; Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). Iris is a framework for state-of-the-art higher-order concurrent separation logics. We use Iris because (1) it allows us to define our logical relations and reason about them at a higher level of abstraction (compared to an explicit model construction); (2) we side-step the well-known type-world-circularity problems (Ahmed, Appel, and Virga, 2002; Ahmed, 2004; Birkedal, Reus, Schwinghammer, Støvring, Thamsborg, and Yang, 2011) involved in defining logical relations for programming languages with higher-order store (since that is already "taken care of" by the model of Iris); and (3) we can leverage the Coq implementation of the Iris base logic (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017) and the Iris Proof Mode (Krebbers, Timany, and Birkedal, 2017) when mechanizing our development in Coq. Indeed, accompanying this paper is a tool for mechanized relational verification of concurrent programs with continuations. The mechanization has been done in Coq and all the results in the paper have been formally verified.

One of the most important features of concurrent separation logics for reasoning about concurrent imperative programs, *e.g.* Dinsdale-Young, Dodds, Gardner, Parkinson, and Vafeiadis (2010), Dinsdale-Young, Birkedal, Gardner, Parkinson, and Yang (2013), Jung, Krebbers, Birkedal, and Dreyer (2016), Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer (2015), Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017), Krebbers, Timany, and Birkedal (2017), Ley-Wild and Nanevski (2013), Nanevski, Ley-Wild, Sergey, and Delbianco (2014), O'Hearn (2007), Sergey, Nanevski, and Banerjee (2015), Svendsen and Birkedal (2014), Turon, Dreyer, and Birkedal (2013), and da Rocha Pinto, Dinsdale-Young, and Gardner (2014), is the support for *modular / local* reasoning. In particular, weakest preconditions and Hoare-triples enable *thread-local* and *context-local* reasoning. Here thread-local means that we can reason about each thread in isolation: when we reason about a particular thread,

we need not explicitly consider interactions from other concurrently executing threads. Similarly, context-local means that when we reason about a particular expression, we need not consider under which evaluation context it is being evaluated. The latter is sometimes codified by the soundness of a proof rule such as the following:

$$\textsc{Hoare-Bind (inadmissible in presence of continuations)}\quad \frac{\{P\}\, e\, \{\Psi\} \qquad \forall w.\ \{\Psi(w)\}\, K[w]\, \{\Phi\}}{\{P\}\, K[e]\, \{\Phi\}}$$

The Hoare-triple $\{P\}\, e\, \{\Psi\}$ intuitively means that, given precondition P, expression e is *safe* and, whenever it reduces to a value v, we are guaranteed that $\Psi(v)$ holds. Intuitively, the above rule expresses that to prove a Hoare triple for an expression e in an evaluation context K, it suffices to prove a property for e in isolation from K, and then show that the desired postcondition Φ can be obtained when substituting a value w satisfying the postcondition Ψ for e into the evaluation context. In a programming language with control operators, *e.g.* `call/cc` and `throw`, the context under which a program is being evaluated is of utmost importance, and thus the above proof rule is *not* sound in general.

Thus, in general, when reasoning about concurrent programs with continuations in a concurrent separation logic, we cannot use context-local reasoning. Hence as part of this work, we develop new non-context-local proof rules for Hoare triples. Those are somewhat more elaborate to use than the standard context-local rules, but that is the price we have to pay to be able to reason in general about non-local control flow. We define our logical relation in terms of Hoare triples, following earlier work (Krebbers, Timany, and Birkedal, 2017; Turon, Dreyer, and Birkedal, 2013), and thus we use the non-context-local proof rules for establishing contextual equivalence of concurrent programs with continuations (and also when proving the soundness of the logical relation itself). To simplify reasoning about parts of programs that do not use control operators, we introduce a new notion of *context-local Hoare triples*. They are defined in terms of the non-context-local Hoare triples and therefore we are able to mix and match reasoning steps using (non-context local) Hoare triples and context-local Hoare triples.

Contributions In this paper, we make the following contributions:

- We present a program logic (weakest preconditions and Hoare-triples) for reasoning about programs written in $\mathsf{F}^{\mu,ref}_{conc,cc}$, a programming language with impredicative polymorphism, recursive types, higher-order functions, higher-order store, concurrency and first-class continuations.

- We present context-local weakest-preconditions and Hoare-triples which simplify reasoning about programs without non-local control flow.
- We present a novel logical relations model for $\mathsf{F}^{\mu,ref}_{conc,cc}$.
- We use our logical relations model and context-local reasoning to prove equivalence of two simple web server implementations: a continuation-based one and a state-storing one.
- We further use our logical relations model to prove correctness of Friedman and Haynes (1985) encoding of continuations by means of one-shot continuations in a concurrent programming language.
- We have developed a fully formalized tool for mechanized interactive relational verification of concurrent programs with continuations. Our tool is developed on top of Iris, a state-of-the-art program logic framework, and we have used it to mechanize all of our contributions in the Coq proof assistant.

Before we begin with the more technical development, we show the essential parts of a continuation-based and a store-based server, which we later on show are contextually equivalent.

7.1.1 Two Servers

Figure 7.1 shows implementations of two handlers mimicking rudimentary web servers. We use an ML-like syntax for the sake of brevity and legibility, but all our example programs can be written in the syntax of our programming language, $\mathsf{F}^{\mu,ref}_{conc,cc}$, and that is indeed what we have done in our Coq formalization. Given a connection, `serverConnT`, a pair of functions for reading and writing, each handler will sum up the numbers given by the client so far, and return the sum back to the client together with a *resumption id*. The client may chose to resume an old computation by giving a new number along with a resumption id or simply make a request to start a computation by giving the first number in the sum to be computed.

`handler1` is a store-based implementation, which stores each state (the sum so far) together with a resumption id in a table. `handler2` is a continuation-based implementation. It simply implements a loop in a fashion as though the user interaction is taking place over a terminal rather than between a client and a server. This loop prints the sum so far and subsequently reads from the client. The operation of reading a value from the client is implemented using the `call/cc` command to capture the current continuation. The captured

```
let handler1 :  ServerConnT -> 1 =
  let tb = newTable () in
  fun (cn : ServerConnT) ->
    let (reader, writer) = cn in
    match reader () with
      (Some cid, n) ->
      begin
        match get tb cid with
          None -> () (* unknown resumption id! *)
        | Some sum ->
          writer (result (sum + n));
          writer (resumptionid (associate tb (sum + n)));
          abort
      end
    | (None, n) ->
      writer (result n);
      let cid = associate tb n
      in writer (resumptionid cid)
```

```
let read_client tb writer =
  callcc (k. writer (resumptionid (associate tb k));
              abort)

let handler2 : ServerConnT -> 1 =
  let tb = newTable () in
  fun (cn : ServerConnT) ->
    let (reader, writer) = cn in
    match reader () with
      (Some cid, n) ->
      begin
        match get tb cid with
        | None -> () (* unknown resumption id! *)
        | Some k -> throw (n, reader, writer) to k
      end
    | (None, n) ->
      let rec loop m reader writer =
        writer (result m);
        let (v, reader, writer) = read_client tb writer in
        loop (m + v) reader writer
      in loop n reader writer
```

Figure 7.1: Two server handlers: one storing the state of the server explicitly (top) and one storing the continuation (bottom)

continuation is associated to a resumption id which is given to the client, so that it may continue the computation by providing a new value to be added to the current sum along with the resumption id in question. Note that since after "reading from the client", we will be communicating with the client on a different connection, we need the `read_client` function to return the new connection as well as the "read value".[1] See Queinnec (2004) and Krishnamurthi, Hopkins, McCarthy, Graunke, Pettyjohn, and Felleisen (2007) for more details on how these kinds of web servers are implemented and used.

One important advantage of using the continuation-based server implementation strategy is scalability. In our rudimentary example in Figure 7.1, the state of the server for each client is primitively simple: the current sum! In general, the state of the web server can be fairly complex. For instance, for a simple web shop, the state of the server for each client includes at the very least: authentication, the contents of the shopping basket, and the information corresponding to every completed stage of ordering: shipping address, shipping method, payment method, payment having been processed or not, etc. Indeed, the complexity of the state is in some cases the main reason for functionality flaws in web applications (Krishnamurthi, Hopkins, McCarthy, Graunke, Pettyjohn, and Felleisen, 2007).

A store-based server implementation for such a web shop could be implemented as follows:

```
...
 match reader () with
    (Some cid, req) ->
    begin
      match get cid with
        None -> () (* unknown resumption id! *)
      | Some state -> resumeShopping req state
    end
  | (None, req) -> startShopping req
...
```

Here the `startShopping` function initializes a session and displays the home page. When given the stored state and the current request, the `resumeShopping` function has to resume the operation by determining what needs to be done based on the given state:

```
let resumeShopping state =
  if not state.authenticated then authenticate ()
  else if state.address = None then ...
  else if  ...
```

[1] The command `abort` is the command that ends the program (thread) and can be written in our programming language as `throw () to -` where `-` is the empty evaluation context.

On the other hand, the continuation-based implementation can be implemented as follows:

```
...
 match reader () with
    (Some cid, req) ->
    begin
      match get cid with
        None -> () (* unknown resumption id! *)
      | Some k -> throw (req, reader, writer) to k
    end
  | (None, req) -> startShoppingCC req
...
```

Here the function `startShoppingCC` is implemented in a much more direct style, as though it is interfacing with the user on a console:

```
let startShoppinCC req =
  let (credentials, reader, writer) = authenticate () in
  let (shippingaddr, reader, writer) = getShippingAddress ()
  in ...
```

This works, because the continuation keeps track of the state (!) and thus, we need not case analyze explicitly on stored state. Note also that in the continuation-based reader, when a valid resumption id is provided, the program simply uses the stored continuation.

A server program using either of our two handlers could be implemented as follows:

```
let rec serve (listener : 1 -> ServerConnT)
   (handler : ServerConnT -> 1) : 1 =
 let v = listener () in
 fork {handler v};  serve listener handler
```

This server program accepts a listener (a function returning a connection) and a handler. It loops, waiting for connections. For each connection it creates a new thread and hands the connection over to its handler. This server program can be applied to any proper listener and either of the handlers depicted in Figure 7.1.

The following is an example of a client program that can interface with the above server (when instantiated with either handler):

```
let send_receive cid number =
  let (reader1, writer1, reader2, writer2) =
     newConnection ()
  in contact_server (reader2, writer1);
     writer1 (cid, number); reader2 ()

```

```
7  let client () =
8    let (cid1, ans1) = send_receive None 10 in
9    (* ans1 = 10 *)
10   let (cid2, ans2) = send_receive (Some cid1) 5 in
11   (* ans2 = 15 *)
12   let (cid3, ans3) = send_receive (None) 17 in
13   (* ans3 = 17 *)
14   let (cid4, ans4) = send_receive (Some cid3) 9 in
15   (* ans4 = 26 *)
16   let (cid5, ans5) = send_receive (Some cid1) 19 in
17   (* ans5 = 29 *)
18   let (cid6, ans6) = send_receive (Some cid2) 17 in
19   (* ans6 = 32 *)
20   ()
```

It shows that a client can indeed go back and forth on the state, e.g., by pressing the back button in the browser. For instance, the resumption id `cid1` is resumed twice, once on line 10 and once on line 16, with a few interactions there in between. In this program the function `send_receive` creates a new connection, sends it to the server (establishing a connection to the server) and subsequently makes the request and retrieves the response from the server and returns it.

7.2 The language: $\mathsf{F}^{\mu,ref}_{conc,cc}$

The language that we consider in this paper, $\mathsf{F}^{\mu,ref}_{conc,cc}$, is a typed lambda calculus with a standard call-by-value small-step operational semantics. It features impredicative polymorphism, recursive types, higher-order mutable references, fine-grained concurrency and first-class continuations. The types of $\mathsf{F}^{\mu,ref}_{conc,cc}$ are as follows:

$$\tau ::= X \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \tau \to \tau \mid \forall X.\ \tau \mid \mu X.\ \tau \mid \tau \times \tau \mid \tau + \tau \mid \texttt{ref}(\tau) \mid \texttt{cont}(\tau)$$

The type $\texttt{ref}(\tau)$ is the type of references with contents of type τ and $\texttt{cont}(\tau)$ is the type of continuations that can be resumed by throwing them a value of type τ.

The syntax for expressions and values is:

$$\begin{aligned} e ::= {} & x \mid () \mid \texttt{true} \mid \texttt{false} \mid n \mid e \circledcirc e \mid \texttt{rec}\, f(x) = e \mid e\ e \mid \Lambda\, e \mid e\ \_ \mid \texttt{fold}\, e \\ & \mid \texttt{unfold}\, e \mid (e, e) \mid \pi_i\, e \mid \texttt{inj}_i\, e \mid \texttt{match}\, e\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} \mid \ell \\ & \mid \texttt{ref}(e) \mid\ !e \mid e \leftarrow e \mid \texttt{CAS}(e, e, e) \mid \texttt{fork}\ \{e\} \mid \texttt{cont}(K) \mid \texttt{call/cc}\,(x.\, e) \end{aligned}$$

$$\mid \texttt{throw}\, e\, \texttt{to}\, e$$

$$v ::= () \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, v \mid (v, v) \mid \texttt{inj}_i\, v \mid \ell$$

$$\mid \texttt{cont}(K)$$

We write n for natural numbers and the symbol $\circledcirc$ stands for binary operations on natural numbers (both basic arithmetic operations and basic comparison operations). We consider both recursive functions $\texttt{rec}\, f(x) = e$ and polymorphic type abstractions $\Lambda\, e$ value. We write $e\ \_$ for type level application (e is a polymorphic expression). We use `fold` and `unfold` to fold and unfold elements of recursive types. Memory locations loc are values of reference types. The expression $!\, e$ reads the memory location e evaluates to, and $e \leftarrow e'$ is an assignment of the value computed by e' to the memory location computed by e. The expression $\texttt{fork}\ \{e\}$ is for forking off a new thread to compute e and we write $\texttt{CAS}(e, e', e'')$ for the compare-and-set operation. A continuation, $\texttt{cont}(K)$, is essentially a suspended evaluation context (see the operational semantics below).

Evaluation contexts of $\mathsf{F}^{\mu,ref}_{conc,cc}$ are as follows:

$$K ::= -\ \mid K\ e \mid v\ K \mid K\ \_ \mid \texttt{fold}\, K \mid \texttt{unfold}\, K \mid \texttt{if}\, K\, \texttt{then}\, e\, \texttt{else}\, e \mid (K, e)$$

$$\mid (v, K) \mid \pi_i\, K \mid \texttt{inj}_i\, K \mid \texttt{match}\, K\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} \mid \texttt{ref}(K) \mid\ !\, K$$

$$\mid K \leftarrow e \mid v \leftarrow K \mid \texttt{CAS}(K, e, e) \mid \texttt{CAS}(v, K, e) \mid \texttt{CAS}(v, v, K)$$

$$\mid \texttt{throw}\, K\, \texttt{to}\, e \mid \texttt{throw}\, v\, \texttt{to}\, K$$

The evaluation context $-$ is the empty evaluation context.

7.2.1 Typing

An excerpt of the typing rules is depicted in Figure 7.2. The context $\Xi = \alpha_1, \ldots, \alpha_n$ is a list of distinct type variables and the context $\Gamma = x_1 : \tau_1, \ldots x_n; \tau_n$ assigns types to program variables. The notation $K : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi \mid \Gamma; \tau')$ means that the evaluation context K satisfies the property that $\Xi \mid \Gamma \vdash K[e] : \tau'$ holds whenever $\Xi \mid \Gamma \vdash e : \tau$ does.

7.2.2 Operational semantics

We define the call-by-value small-step operational semantics of $\mathsf{F}^{\mu,ref}_{conc,cc}$ in two stages. We first define a head-step relation $\rightarrow_K$. Here, K is the context under

$$\text{T-Var}\ \frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau} \qquad \text{T-Unit}\ \frac{}{\Xi \mid \Gamma \vdash () : 1} \qquad \text{T-Nat}\ \frac{}{\Xi \mid \Gamma \vdash n : \mathbb{N}} \qquad \text{T-TLam}\ \frac{\Xi, X \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall X.\, \tau}$$

$$\text{T-Rec}\ \frac{\Xi \mid \Gamma, x : \tau, f : \tau \to \tau' \vdash e : \tau'}{\Xi \mid \Gamma \vdash \ \mid \mathtt{rec}\, f(x) = e : \tau \to \tau'} \qquad \text{T-TApp}\ \frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]}$$

$$\text{T-Fold}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau/X]}{\Xi \mid \Gamma \vdash \mathtt{fold}\, e : \mu X.\, \tau} \qquad \text{T-UnFold}\ \frac{\Xi \mid \Gamma \vdash e : \mu X.\, \tau}{\Xi \mid \Gamma \vdash \mathtt{unfold}\, e : \tau[\mu X.\, \tau/X]}$$

$$\text{T-Ref}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathtt{ref}(e) : \mathtt{ref}(\tau)} \qquad \text{T-DeRef}\ \frac{\Xi \mid \Gamma \vdash e : \mathtt{ref}(\tau)}{\Xi \mid \Gamma \vdash\, !\, e : \tau} \qquad \text{T-Fork}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathtt{fork}\ \{e\} : 1}$$

$$\text{T-Cont}\ \frac{K : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi \mid \Gamma; \tau')}{\Xi \mid \Gamma \vdash \mathtt{cont}(K) : \mathtt{cont}(\tau)} \qquad \text{T-Assign}\ \frac{\Xi \mid \Gamma \vdash e : \mathtt{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : 1}$$

$$\text{T-CAS}\ \frac{\Xi \mid \Gamma \vdash e_1 : \mathtt{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau \qquad \Xi \mid \Gamma \vdash e_3 : \tau}{\Xi \mid \Gamma \vdash \mathtt{CAS}(e_1, e_2, e_3) : 1}$$

$$\text{T-Call/cc}\ \frac{\Xi \mid \Gamma, x : \mathtt{cont}(\tau) \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathtt{call/cc}\ (x.\, e) : \tau} \qquad \text{T-Throw}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \mid \Gamma \vdash e' : \mathtt{cont}(\tau)}{\Xi \mid \Gamma \vdash \mathtt{throw}\, e\, \mathtt{to}\, e' : \tau'}$$

Figure 7.2: An excerpt of the typing rules

$$((\texttt{rec}\, f(x) = e)\ v, \sigma) \to_K (e[v, (\texttt{rec}\, f(x) = e)/x, f], \sigma)$$

$$((\Lambda\, e)\ \_, \sigma) \to_K (e, \sigma) \qquad (\texttt{unfold}\,(\texttt{fold}\, v), \sigma) \to_K (v, \sigma)$$

$$(\texttt{if}\,\texttt{true}\,\texttt{then}\, e_2\,\texttt{else}\, e_3, \sigma) \to_K (e_2, \sigma) \qquad (\pi_1\,(v_1, v_2), \sigma) \to_K (v_1, \sigma)$$

$$\frac{\ell \notin \mathrm{dom}(\sigma)}{(\texttt{ref}(v), \sigma) \to_K (\ell, \sigma \uplus \{\ell \mapsto v\})} \qquad \frac{\sigma = \sigma' \uplus \{\ell \mapsto v'\}}{(\ell \leftarrow v, \sigma) \to_K ((), \sigma' \uplus \{\ell \mapsto v\})}$$

$$\frac{v = \sigma(\ell)}{(!\,\ell, \sigma) \to_K (v, \sigma)} \qquad \frac{\sigma = \sigma' \uplus \{\ell \mapsto v\}}{(\texttt{CAS}(\ell, v, v'), \sigma) \to_K (\texttt{true}, \sigma' \uplus \{\ell \mapsto v'\})}$$

$$\frac{\sigma = \sigma' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\texttt{CAS}(\ell, v, v'), \sigma) \to_K (\texttt{false}, \sigma)} \qquad (\texttt{call/cc}\,(x.\, e), \sigma) \to_K (e[\texttt{cont}(K)/x], \sigma)$$

Figure 7.3: An excerpt of the head-reduction rules

which the head step is being performed. Based on this, we define the operational semantics of programs by what we call *the thread-pool step relation* $\to$. A thread pool reduces by making a head reduction step in one of the threads, by forking off a new thread, or by resuming a captured continuation:

$$\frac{(e, \sigma) \to_K (e', \sigma')}{(\vec{e}_1, K[\,e\,], \vec{e}_2; \sigma) \to (\vec{e}_1, K[\,e'\,], \vec{e}_2; \sigma')}$$

$$(\vec{e}_1, K[\,\texttt{fork}\,\{e\}\,], \vec{e}_2; \sigma) \to (\vec{e}_1, K[\,()\,], \vec{e}_2, e; \sigma)$$

$$(\vec{e}_1, K[\,\texttt{throw}\, v\,\texttt{to}\,\texttt{cont}(K')\,], \vec{e}_2; \sigma) \to (\vec{e}_1, K'[\,v\,], \vec{e}_2; \sigma)$$

Here, σ is the physical state of the program, *i.e.*, the program heap, which is a finite partial map from memory locations to values. An excerpt of the head-step relation is given in Figure 7.3. Notice that the head-step for `call/cc` captures the continuation that is the index of the head-step relation.

Contextual refinement/equivalence A program e contextually refines a program e' of type τ if both programs have type τ and no *well-typed* context (a closed top-level program with a hole) can distinguish a situation where e' is replaced by e.

$$\Xi \mid \Gamma \vdash e \leq_{\text{ctx}} e' : \tau \triangleq \Xi \mid \Gamma \vdash e : \tau \wedge \Xi \mid \Gamma \vdash e' : \tau \wedge$$

$$\forall K.\ K : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; 1) \wedge K[e] \Downarrow \Rightarrow K[e'] \Downarrow$$

where

$$e \Downarrow \triangleq \exists v, \vec{e}, \sigma.\ (e; \emptyset) \rightarrow^* (v, \vec{e}; \sigma)$$

The intuitive explanation above for contextual refinement is the reason why in a contextual refinement $e \leq_{\text{ctx}} e$ or in a logical relatedness relation $e \leq_{\log} e'$, usually, the program on the left hand side, e, is referred to as the implementation side and the program on the right hand side, e', is referred to as the specification side.

Two programs are contextually equivalent, if each contextually refines the other:

$$\Xi \mid \Gamma \vdash e \approx_{\text{ctx}} e' : \tau \triangleq \Xi \mid \Gamma \vdash e \leq_{\text{ctx}} e' : \tau \wedge \Xi \mid \Gamma \vdash e' \leq_{\text{ctx}} e : \tau$$

7.3 Logical relations

It is challenging to construct logical relations for languages with higher-order store because of the so-called type-world circularity (Ahmed, Appel, and Virga, 2002; Ahmed, 2004; Birkedal, Reus, Schwinghammer, Støvring, Thamsborg, and Yang, 2011). The logic of Iris is rich enough to allow for a direct inductive specification of the logical relations for programming languages with advanced features such as higher-order references, recursive types, and concurrency (Krebbers, Timany, and Birkedal, 2017; Krogh-Jespersen, Svendsen, and Birkedal, 2017; Timany, Stefanesco, Krogh-Jespersen, and Birkedal, 2018).

7.3.1 An Iris primer

Iris (Jung, Krebbers, Birkedal, and Dreyer, 2016; Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer, 2015; Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017) is a state-of-the-art higher-order concurrent separation logic designed for verification of programs.

In Iris one can quantify over the Iris types κ:

$$\kappa ::= 1 \mid \kappa \times \kappa \mid \kappa \to \kappa \mid \mathit{Ectx} \mid \mathit{Var} \mid \mathit{Expr} \mid \mathit{Val} \mid \mathbb{N} \mid \mathbb{B} \mid \kappa \xrightarrow{\text{fin}} \kappa \mid \text{finset}(\kappa) \mid \mathit{Monoid}$$

$$\mid \mathit{Names} \mid \mathit{iProp} \mid \ldots$$

Here *Ectx*, *Var*, *Expr* and *Val* are Iris types for evaluation contexts, variables, expressions and values of $\mathsf{F}^{\mu,ref}_{conc,cc}$. Natural numbers, $\mathbb{N}$, and Booleans $\mathbb{B}$ are also included among the base types of Iris. Iris also features partial maps with finite support $\kappa \xrightarrow{\text{fin}} \kappa$ and finite sets, $\text{finset}(\kappa)$. Resources in Iris are represented using partial commutative monoids, *Monoid*, and instances of resources are named using so-called ghost-names *Names*. Finally, and most importantly, there is a type of Iris propositions *iProp*. The grammar for Iris propositions is as follows:

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\, \Phi \mid \exists x : \kappa.\, \Phi$$

$$\mid \triangleright P \mid \mu r.P \mid \Box P \mid \mathsf{wp}\, e\, \{x.\ P\} \mid \{P\}\, e\, \{x.\ Q\} \mid \Rightarrow\!\!\!\Rightarrow P \mid \boxed{P}^{\mathcal{N}} \mid \ldots$$

Here, $\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$ are the standard higher-order logic connectives. The predicates Φ are Iris predicates, i.e., terms of type $\kappa \to \mathit{iProp}$.

The connective $*$ is the separating conjunction. Intuitively, $P * Q$ holds if resources owned can be split into two *disjoint* pieces such that one satisfies P and the other Q. The magic wand connective $P \mathrel{-\!\!*} Q$ is satisfied by resources such that when these resources are combined with some resource satisfying P the resulting resources would satisfy Q.

The $\triangleright$ modality, pronounced "later" is a modality that intuitively corresponds to some abstract form of step-indexing (Appel and McAllester, 2001; Appel, Melliès, Richards, and Vouillon, 2007; Dreyer, Ahmed, and Birkedal, 2011). Intuitively, $\triangleright P$ holds if P holds one step into the future. Iris has support for taking fixed points of *guarded* propositions, $\mu r.P$. This fixed point can only be defined if all occurrences of r in P are guarded, *i.e.*, appear under a $\triangleright$ modality. We use guarded fixed points for defining the interpretation of recursive types in $\mathsf{F}^{\mu,ref}_{conc,cc}$. For any proposition P we have $P \vdash \triangleright P$.

When the modality $\Box$ is applied to a proposition P, the non-duplicable resources in P are forgotten, and thus $\Box P$ is "persistent." In general, we say that a proposition P is *persistent* if $P \dashv\vdash \Box P$ (where $\dashv\vdash$ is the logical equivalence of Iris propositions). A key property of persistent propositions is that they are duplicable: $P \dashv\vdash P * P$. The type system of $\mathsf{F}^{\mu,ref}_{conc,cc}$ is not a sub-structural type system and variables (in the typing environment) may be used multiple times. Therefore when we interpret types as logical relations in Iris, then those relations should be duplicable. We use the persistence modality $\Box$ to ensure this.

Iris facilitates specification and verification of programs by means of weakest-preconditions $\mathsf{wp}\, e\, \{v.\ P\}$, which intuitively hold whenever e is *safe* and,

moreover, whenever e terminates with a resulting value v, then $P[v/x]$ holds. When x does not appear in P we write $\mathsf{wp}\, e\, \{x.\, P\}$ as $\mathsf{wp}\, e\, \{P\}$. Also, we sometimes write $\mathsf{wp}\, e\, \{\Phi\}$ for $\mathsf{wp}\, e\, \{x.\, \Phi(x)\}$

In Iris, Hoare triples are defined in terms of weakest preconditions like this: $\{P\}\, e\, \{v.\, Q\} \triangleq \Box\, (P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\, Q\})$. Note that the $\Box$ modality ensures that the Hoare triples are persistent and hence duplicable (in separation logic jargon, Hoare triples should just express "knowledge" and not claim ownership of any resources).

A key feature of Iris (as for other concurrency logics) is that specification and verification is done *thread-locally*: the weakest precondition only describes properties of execution of a single thread. Concurrent interactions are abstracted and reasoned about in terms of resources (rather than by explicit reasoning about interleavings). For programming languages that do not include continuations or other forms of non-local control flow, the weakest precondition is not only thread-local, but also what we may call *context-local*. Context-local means that to reason about an expression in an evaluation context, it suffices to reason about the expression in isolation, and then separately about what the context does to the resulting value. This form of context-locality is formally expressed by the soundness of the following bind rule

$$\textsc{inadmissible-bind}\quad \frac{\mathsf{wp}\, e\, \{v.\, \mathsf{wp}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}\, K[e]\, \{\Phi\}}$$

Clearly, this rule is not sound when expressions include `call/cc` (since `call/cc` captures the context its behaviour depends on the context). See Appendix F for a proof of inadmissibility of this rule.

Thus, for reasoning about $\mathsf{F}^{\mu,ref}_{conc,cc}$ we cannot use the "standard" Iris rules (Jung, Krebbers, Birkedal, and Dreyer, 2016; Jung, Swasey, Sieczkowski, Svendsen, Turon, Birkedal, and Dreyer, 2015; Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017; Krebbers, Timany, and Birkedal, 2017) for weakest preconditions. Instead, we use new rules such as the following — the difference from the standard rules is that our new rules include an explicit context K (earlier, such rules could be derived using the bind rule, but that is not sound in general so we cannot do that). Note that the context is used in the rules CALLCC-WP and THROW-WP for `call/cc` and `throw`. These two rules directly reflect the operational semantics of `call/cc` and `throw`.

$$\textsc{fst-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K[v]\, \{\Phi\}}{\mathsf{wp}\, K[\pi_1\, (v, w)]\, \{\Phi\}} \qquad \textsc{if-true-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K[e]\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{if true then}\, e\, \texttt{else}\, e']\, \{\Phi\}}$$

$$\text{REC-WP}\quad \frac{\triangleright\, \mathsf{wp}\, K[e[\texttt{rec}\, f(x) = e, v/f, x]]\, \{\Phi\}}{\mathsf{wp}\, K[(\texttt{rec}\, f(x) = e)\ v]\, \{\Phi\}} \qquad \text{CALLCC-WP}\quad \frac{\triangleright\, \mathsf{wp}\, K[e[\texttt{cont}(K)/x]]\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{call/cc}\, (x.\, e)]\, \{\Phi\}}$$

$$\text{THROW-WP}\quad \frac{\triangleright\, \mathsf{wp}\, K'[v]\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{throw}\, v\, \texttt{to}\, \texttt{cont}(K')]\, \{\Phi\}}$$

In summa, for $\mathsf{F}^{\mu,ref}_{conc,cc}$ *we use new non-context-local rules for reasoning about weakest preconditions, and the non-context-local rules allow us to reason about* `call/cc` *and* `throw`*.*

Because of the explicit context K, the non-context-local rules for weakest preconditions are somewhat more elaborate to use than the corresponding context-local rules. However, that is the price we have to pay to be able to reason in general about non-local control flow. In Section 7.4 we will see how we can still recover a form of context-local weakest precondition for reasoning about those parts of the program that do not use non-local control flow.

In the rules above, the antecedent is only required to hold a step of computation later ($\triangleright$) — that is because these rules correspond to expressions performing a reduction step.

The update modality $\Rrightarrow$ accounts for updating (allocation, deallocation and mutation) of resources.[2] Intuitively, $\Rrightarrow P$ is satisfied by resources that can be updated to new resources for which P holds. For any proposition P, we have that $P \vdash \Rrightarrow P$. If P holds, then resources can be updated (trivially) so as to have that P holds. The update modality is also idempotent, $\Rrightarrow\Rrightarrow P \dashv\vdash \Rrightarrow P$. We write $P \Rrightarrow\!\!\!\!\ast\ Q$ as shorthand for $P \mathrel{-\!\!*} \Rrightarrow Q$. Crucially, resources can be updated throughout a proof of weakest preconditions:

$$\Rrightarrow \mathsf{wp}\, e\, \{\Phi\} \dashv\vdash \mathsf{wp}\, e\, \{\Phi\} \dashv\vdash \mathsf{wp}\, e\, \{v.\ \Rrightarrow \Phi(v)\}$$

Iris features invariants $\boxed{P}^{\mathcal{N}}$ for enforcing concurrent protocols. Each invariant $\boxed{P}^{\mathcal{N}}$ has a name, $\mathcal{N}$, associated to it. Names are used to keep track of which invariants are open.[3] Intuitively, $\boxed{P}^{\mathcal{N}}$ states that P always holds. The following

[2] This modality is called the fancy update modality in Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017).

[3] Officially in Iris, the update modality is in fact annotated with so-called masks (sets of invariant names), which are used to ensure that invariants are not re-opened. For simplicity, we do not include masks in this paper.

rules govern invariants.

$$\text{INV-ALLOC}\ \frac{\triangleright P}{\Rightarrow \boxed{P}^{\mathcal{N}}} \qquad \text{INV-OPEN-WP}\ \frac{\boxed{R}^{\mathcal{N}} \qquad (\triangleright R) \twoheadrightarrow \mathsf{wp}\, e\, \{y.\ (\triangleright R) * \mathsf{wp}\, K[y]\, \{x.\ Q\}\} \qquad e \text{ is atomic}}{\mathsf{wp}\, K[e]\, \{x.\ Q\}}$$

These rules say that invariants can always be allocated by giving up the resources being protected by the invariant and they can be kept opened only during execution of *physically atomic* operations. Iris invariants are impredicative, *i.e.*, they can state P holds invariantly for any proposition P, including invariants. This is why the later operator is used as a guard to avoid self-referential paradoxes (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). Invariants essentially express the knowledge that some proposition holds invariantly. Hence, invariants are always persistent, *i.e.*, $\boxed{P}^{\mathcal{N}} \dashv\vdash \Box \boxed{P}^{\mathcal{N}}$.

7.3.2 Resources used in defining logical relations

We need some resources in order to define our logical relations in Iris. We need resources for representing memory locations of the implementation side, the memory locations of the specification side and the expression being evaluated on the specification side. These resources are written as follows:

- $\ell \mapsto_i v$: memory location ℓ contains value v on the implementation side.
- $\ell \mapsto_s v$: memory location ℓ contains value v on the specification side.
- $j \Rightarrow e$: the thread j on the specification is about to execute e.

These resources are defined using more primitive resources in Iris, but we omit such details here. What is important is that we can use these resources to reason about programs. In particular, we can derive the following rules (and similarly for other basic expressions) for weakest preconditions and for execution on the specification side.

$$\frac{\forall \ell.\ \ell \mapsto_i v \twoheadrightarrow \mathsf{wp}\, K[\ell]\, \{\Phi\}}{\mathsf{wp}\, K[\mathtt{ref}(v)]\, \{\Phi\}} \qquad \frac{\ell \mapsto_i v \twoheadrightarrow \mathsf{wp}\, K[v]\, \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[!\,\ell]\, \{\Phi\}}$$

$$\frac{\ell \mapsto_i w \twoheadrightarrow \mathsf{wp}\, K[()]\, \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[\ell \leftarrow w]\, \{\Phi\}} \qquad \frac{j \Rightarrow K[\mathtt{ref}(v)]}{\Rightarrow \exists \ell.\ \ell \mapsto_s v * j \Rightarrow K[\ell]}$$

$$\frac{\ell \mapsto_s v \qquad j \Rightarrow K[!\,\ell]}{\Rightarrow \ell \mapsto_s v * j \Rightarrow K[v]} \qquad \frac{\ell \mapsto_s v \qquad j \Rightarrow K[\ell \leftarrow w]}{\Rightarrow \ell \mapsto_s w * j \Rightarrow K[()]}$$

These resources are all exclusive in the sense that:

$$\ell \mapsto_i v * \ell \mapsto_i v' \vdash \bot \qquad \ell \mapsto_s v * \ell \mapsto_s v' \vdash \bot \qquad j \Rightarrow e * j \Rightarrow e' \vdash \bot$$

7.3.3 Logical relations in Iris

Figure 7.4 presents our binary logical relation for $\mathsf{F}^{\mu,ref}_{conc,cc}$. We define the logical relation in several stages. The first thing we define is the relation of observational refinement. Intuitively, an expression e observationally refines an expression e' if, whenever e reduces to a value so does e'. We define this in Iris using magic wand and weakest precondition. The whole formula reads as follows: Assuming that there is some thread j on the specification side that is about to execute e' (represented in Iris by $j \Rightarrow e'$) then, after execution of e, we know that thread j on the specification side has also been executed to some value w.

We then use the notion of observational refinement defined above to define the value relation, the expression relation and the evaluation context relation for each type. In contrast to earlier definitions of logical relations in Iris (Krebbers, Timany, and Birkedal, 2017; Krogh-Jespersen, Svendsen, and Birkedal, 2017; Timany, Stefanesco, Krogh-Jespersen, and Birkedal, 2018), our logical relation is an example of so-called biorthogonal logical relations (Pitts, 2005), also known as top-top closed logical relations. That is, we define two expressions to be related if plugging them into related evaluation contexts results in observationally related expressions. Two evaluation contexts are defined to be related if plugging related values into them results in observationally related expressions.

The value relation interpretation $[\![\Xi \vdash \tau]\!]_\Delta$ of a type in context is defined by induction on τ. Here Δ is an environment mapping type variables in Ξ to Iris relations. For all the non-continuation types, the definition is exactly as in for the language without `call/cc`, see Krebbers, Timany, and Birkedal (2017), and thus we only include the cases for type variables, Booleans and function types in addition to the new case for the continuation types (the full definition can be found in Appendix F).

Two values of the Boolean type $\mathbb{B}$ are related if they are both `true` or they are both `false`. The value relation for functions types $\tau \to \tau'$ expresses that two values of the function type are related if whenever applied to related values of the domain type, τ, the resulting *expressions* are related at the codomain type, τ'. The use of the *persistently modality* here is to make sure that the interpretations are persistent. Finally, the relational interpretation of $\texttt{cont}(\tau)$ expresses that two continuations are related whenever their corresponding evaluation contexts are related at the evaluation context relation for the type in question.

Observational refinement: $\mathcal{O} : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$

$$\mathcal{O}(e, e') \triangleq \forall j.\ j \Mapsto e' \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\exists w.\ j \Mapsto w\}$$

Value interpretation of types: $[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ for $\Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$

$$[\![\Xi \vdash X]\!]_\Delta \triangleq \Delta(X)$$

$$[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v, v') \triangleq v = v' = \mathtt{true} \lor v = v' = \mathtt{false}$$

$$[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') \triangleq \square\left(\forall w, w'.\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta(v\ w, v'\ w')\right)$$

$$[\![\Xi \vdash \mathtt{cont}(\tau)]\!]_\Delta(v, v') \triangleq \exists K, K'.\ v = \mathtt{cont}(K) \land v' = \mathtt{cont}(K') \land \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K')$$

Evaluation context interpretation of types: $\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Ectx} \times \mathit{Ectx} \to \mathit{iProp}$ for $\Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$

$$\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K') \triangleq \forall v, v'.\ [\![\Xi \vdash \tau]\!]_\Delta(v, v') \Rightarrow \mathcal{O}(K[v], K'[v'])$$

Expression interpretation of types: $[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$ for $\Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$

$$[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e, e') \triangleq \forall K, K'.\ \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K') \Rightarrow \mathcal{O}(K[e], K'[e'])$$

Logical relatedness: $\Xi \mid \Gamma \vDash e \le_{\log} e' : \tau : \mathit{iProp}$

$$\Xi \mid \Gamma \vDash e \le_{\log} e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v'}.\ \left(\mathop{\Large *}_{x_i:\tau_i} [\![\Xi \vdash \tau_i]\!]_\Delta(v_i, v'_i)\right) \Rightarrow [\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$

$$\text{if } \ \Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$$

Figure 7.4: An excerpt of the logical relations for $\mathsf{F}^{\mu,ref}_{conc,cc}$

The evaluation context relation $\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta$ relates evaluation contexts K and K' if plugging related values of type τ in them results in observationally related expressions.

The expression relation is the standard biorthogonal expression relation. It states that $\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(e, e')$ holds whenever, for any two related evaluation contexts K and K', the expressions $K[e]$ and $K'[e']$ are observationally related.

The notion of logical relatedness states, as usual for call-by-value languages, that two expressions e and e' are logically related if substituting related values for their free variables results in related expressions.

We can now state and prove the fundamental theorem of logical relations for $\mathsf{F}^{\mu,ref}_{conc,cc}$. The theorem expresses that any well-typed expression is logically related to itself.

Theorem 7.3.1 (Fundamental theorem of logical relations)**.**

$$\Xi \mid \Gamma \vdash e : \tau \Rightarrow \Xi \mid \Gamma \vDash e \leq_{log} e : \tau$$

This theorem is proven by induction on the typing derivation using the basic rules for weakest-preconditions and executions on the specification side.

The above theorem, together with some basic properties of observational refinement, implies the soundness of our logical relations, i.e., that logical relatedness implies contextual refinement:

Theorem 7.3.2 (Soundness of logical relations)**.**

$$\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau \Rightarrow \Xi \mid \Gamma \vDash e \leq_{ctx} e' : \tau$$

Our logical relation is expressed in terms of weakest preconditions and the proofs of the above theorems use the earlier presented proof rules for weakest preconditions. Before turning to applications, we pause to present *context-local weakest preconditions*, which we can use to simplify reasoning about program fragments, which do not use non-local control flow.

7.4 Context-local weakest preconditions (CLWP)

To make it simpler to reason about expressions that do not use non-local control flow, we define a new notion of *context-local weakest precondition*. The definition is given in terms of the earlier weakest precondition, which, as we will explain below, means that we will be able to mix and match reasoning

steps using (non-context local) weakest preconditions and context-local weakest preconditions.

Definition 7.4.1. *The* context-local weakest precondition *of e wrt. Φ is defined as:*

$$\mathit{clwp}\, e\, \{\Phi\} \triangleq \forall K, \Psi.\ (\forall v.\ \Phi(v) \twoheadrightarrow \mathit{wp}\, K[v]\, \{\Psi\}) \twoheadrightarrow \mathit{wp}\, K[e]\, \{\Psi\}$$

Note how the above definition essentially says that $\mathsf{clwp}\, e\, \{\Phi\}$ holds if the bind rule holds for e, which intuitively means that e does not use non-local control flow. Therefore, the bind rule is sound for context-local weakest preconditions:

$$\text{BIND}\quad \frac{\mathsf{clwp}\, e\, \{v.\ \mathsf{clwp}\, K[v]\, \{\Phi\}\}}{\mathsf{clwp}\, K[e]\, \{\Phi\}}$$

Moreover, the "standard" rules for the basic language constructs (excluding `call/cc` and `throw`, of course) can also be derived for context-local weakest preconditions: Here is an excerpt of the rules that we can derive:

$$\text{FST-CLWP}\quad \frac{\triangleright\, \mathsf{clwp}\, v\, \{\Phi\}}{\mathsf{clwp}\, \pi_1\, (v, w)\, \{\Phi\}} \qquad \text{IF-TRUE-CLWP}\quad \frac{\triangleright\, \mathsf{clwp}\, e\, \{\Phi\}}{\mathsf{clwp}\, \texttt{if true then}\, e\, \texttt{else}\, e'\, \{\Phi\}}$$

$$\text{REC-CLWP}\quad \frac{\triangleright\, \mathsf{clwp}\, e[\texttt{rec}\, f(x) = e, v/f, x]\, \{\Phi\}}{\mathsf{clwp}\, (\texttt{rec}\, f(x) = e)\, v\, \{\Phi\}} \qquad \text{ALLOC-CLWP}\quad \mathsf{clwp}\, \texttt{ref}(v)\, \{w.\ \exists \ell. w = \ell * \ell \mapsto_i v\}$$

$$\text{LOAD-CLWP}\quad \frac{\triangleright\, \ell \mapsto_i v}{\mathsf{clwp}\, !\ell\, \{w.\ w = v * \ell \mapsto_i v\}}$$

We can also use invariants during atomic steps of computation while proving context-local weakest preconditions:

$$\text{INV-OPEN-CLWP}\quad \frac{\boxed{R}^{\mathcal{N}} \qquad (\triangleright R) \twoheadrightarrow \mathsf{clwp}\, e\, \{v.\ (\triangleright R) * Q\} \qquad e \ \text{ is atomic}}{\mathsf{clwp}\, e\, \{v.\ Q\}}$$

Now we have both (non-context-local) weakest preconditions and context-local weakest preconditions. What is the upshot of this? The key point is that when we prove correctness / relatedness of programs, then we can use the

simpler context-local weakest preconditions for reasoning about those parts of the program which are context local (do not use `call/cc` or `throw`) and only use the (non-context-local) weakest preconditions for reasoning about those parts that may involve non-local control flow. This fact is expressed formally by the following derivable rule, which establishes a connection between weakest-preconditions and context-local weakest preconditions.

$$\frac{\text{CLWP-WP}\quad \mathsf{clwp}\, e\, \{\Psi\} \qquad \forall v.\ \Psi(v) \mathrel{-\!\!*} \mathsf{wp}\, K[v]\, \{\Phi\}}{\mathsf{wp}\, K[e]\, \{\Phi\}}$$

This rule basically says that if we know that e *context-locally* guarantees postcondition Ψ then we can prove $\mathsf{wp}\, K[e]\, \{\Phi\}$ by assuming that *locally*, under the context K, it will only evaluate to values that satisfy Ψ. Moreover, it guarantees that evaluation of e does not tamper with the evaluation context that we are considering it under.

Similarly to Hoare-triples above, we define context-local Hoare-triples based on context-local weakest preconditions:

$$\{P\}^{\mathsf{cl}}\, e\, \{v.\ Q\} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{clwp}\, e\, \{v.\ Q\}\right)$$

7.5 Case study: server with continuations

In this section we show that the two server implementations discussed in the Introduction, the continuation-based implementation and the state-storing implementation, are contextually equivalent. Note that our server implementations are parameterized on a pair of functions, one for reading requests from the client and one for writing to the client. The idea is that these functions are an abstraction of a TCP connection and thus the contextual equivalence can be understood as showing that clients cannot distinguish between the two implementations.

The crux of the proof of contextual equivalence is proving that the two handlers in Figure 7.1 are contextually equivalent. Both of these handlers start out by establishing an empty table for storing their resumptions. In the state-storing implementation, the table is used to store the state (the sum so far), while in the continuation-based implementation, the table stores the continuation. After creating the tables, both implementations return functions which are the actual handlers. These functions internally use their respective tables to store and

look-up resumptions. The table implementation itself is straightforward and thus omitted. It uses a spin lock (omitted) for synchronization. Since the table and the lock do not make use of `call/cc` and `throw`, we employ context-local weakest preconditions to give relational specifications for them. Hence we can reason about the table and lock implementations in the way we usually do in Iris for concurrent programs without continuations. When proving relatedness of the handlers, which we do using (non context-local) weakest preconditions, the CLWP-WP rule allows us to make use of the context-local relational specifications of the table and lock.

In the rest of this section we will first present and discuss our relational specifications for the table and the lock. After that we will discuss the logical relatedness of the two handlers. Our relational specifications for the table and the lock are stronger than the specifications one usually encounters in the literature. We need this strengthening because the continuations stored in the table refers to the table itself in the continuation-based implementation. This is, fundamentally, also the reason why, although the table code is identical in both handlers, we cannot use the fundamental theorem of logical relations to conclude that they are related in a sufficiently strong way.

7.5.1 Relational spec for the table and the lock

We now discuss the relational specifications for the tables and the locks that they use for synchronization. All the reasoning in this subsection is context-local, using the primitive rules for context-local weakest preconditions. The table specifications are used in the proof of relatedness of the handlers, which we discuss in the following subsection.

The essence of relating the tables on both sides (specification side and implementation side) is simple. The contents of new tables are related (as they are both empty) and we only store values that are suitably related. Hence, when looking the table up we are guaranteed to receive related values, if any. This is formally captured in the following relational specifications:

$$\{j \Rrightarrow K[\mathbf{newTable}\ ()]\}^{\mathsf{cl}}\ \mathbf{newTable}\ ()$$

$$\{v.\ \exists v'.\ j \Rrightarrow K[v'] * \forall \Phi.\ \Rrightarrow \exists \gamma.\ \mathit{relTables}(v, v', \gamma, \Phi)\}$$

$$\{relTables(tb, tb', \gamma, \Phi) * j \Rrightarrow K[\mathbf{get}\ tb'\ n]\}^{\mathsf{cl}}\ \mathbf{get}\ tb\ n$$

$$\left\{\begin{array}{l} v.\ \exists v'.\ j \Rrightarrow K[v'] * \big(v = v' = None \vee \\ \quad (\exists w, w'\ v = Some(w) \wedge v' = Some(w') * \Phi(v, v'))\big) \end{array}\right\}$$

$$\left\{\begin{array}{l} relTables(tb, tb', \gamma, \Phi) * \Phi(v, v') \\ * j \Rrightarrow K[\mathbf{associate}\ tb'\ v] \end{array}\right\}^{\mathsf{cl}}$$

$$\mathbf{associate}\ tb\ v$$

$$\{v.\ \exists n.\ v = n * j \Rrightarrow K[n]\}$$

The specifications for **get** and **associate** are exactly as we explained above. The most important part of this spec is the persistent proposition $relTables(v, v', \gamma, \Phi)$ which intuitively says that the tables v and v' have contents that are pair-wise related by the binary predicate Φ. The name γ for ghost resources is used for synchronization purposes. The specification of **newTable** is *stronger* than usual in that it guarantees that for any user picked predicate we can obtain that the two tables are related. Contrast this with the weaker standard style specification

$$\forall \Phi.\ \{j \Rrightarrow K[\mathbf{newTable}\ ()]\}^{\mathsf{cl}}\ \mathbf{newTable}\ () \qquad \text{(weaker standard spec)}$$
$$\{v.\ \exists v'.\ j \Rrightarrow K[v'] * \exists \gamma.\ relTables(v, v', \gamma, \Phi)\}$$

which quantifies over Φ outside the whole triple.

Notice that with our stronger specification we can refer to the tables themselves in the predicate Φ that we pick for relating the contents, whereas in the (weaker standard spec) specification one has to pick this relation beforehand, and hence one cannot refer to the tables v and v' because they have not been created yet!

The predicate $relTables(tb, tb', \gamma, \Phi)$ is defined in terms of the *relLocks* predicate, which pertains to the relational specification of spin locks given below.

$$relTables(tb, tb', \gamma, \Phi) \triangleq relLocks(tb.lock, tb'.lock, \gamma, P_\Phi)$$

$$P_\Phi \triangleq \exists ls.\ contents(tb, map\ \pi_1\ \ ls) * contents(tb', map\ \pi_2\ \ ls) * \mathop{\Large *}_{(x,x') \in ls} \Phi(x, x')$$

Here $tb.lock$ is the lock associated with the table tb. The proposition P_Φ above simply states that the there is a list of pairs of values, which are pairwise related by Φ and, moreover, that the first projections of these pairs are stored in the implementation side table and the second projections of these pairs are stored in the specification side table. The *contents* predicate simply specifies that the index of an element in the table is its index in the list.

Relational spec for the spin lock We use the following relational specification for relating the locks used on the implementation and the specification side.

$$\{j \Mapsto K[\mathbf{newlock}\ ()]\}^{\mathsf{cl}}$$

$$\quad \mathbf{newlock}\ ()$$

$$\{v.\ \exists v'.\ j \Mapsto K[v'] * \forall P.P \Rrightarrow \exists \gamma.\ relLocks(v, v', \gamma, P)\}$$

$$\{relLocks(v, v', \gamma, P) * j \Mapsto K[\mathbf{acquire}\ v']\}^{\mathsf{cl}}$$

$$\quad \mathbf{acquire}\ v$$

$$\{\_.\ j \Mapsto K[()] * locked(\gamma) * P\}$$

$$\{relLocks(v, v', \gamma, P) * P * locked(\gamma) * j \Mapsto K[\mathbf{release}\ v']\}^{\mathsf{cl}}$$

$$\quad \mathbf{release}\ v$$

$$\{\_.\ j \Mapsto K[()]\}$$

The specification captures that whenever we acquire the lock on the implementation side, the lock on the specification side is free and can be acquired. We need this for showing contextual refinements because if the implementation side converges, then we need to show that so does the specification side and the acquire operation is potentially non-terminating. This also means that whenever we release the lock on the implementation side, the lock on the specification side is also released.

Our relational lock specification is also a bit stronger than usual, (cf. the quantification over P in the **newlock** specification), because we use the lock specification when proving the relational specification for tables described above.

The persistent proposition $\mathit{relLocks}(v, v', \gamma, P)$ states that v is a lock protecting two things: resources P *and* the fact that v' is not acquired. The proposition $\mathit{locked}(\gamma)$ states that both of the locks associated to γ are currently acquired.

7.5.2 Proving equivalence of handlers

We devote the reset of this section to discussing the main result of this section: proving relatedness of handlers in Figure 7.1.

Theorem 7.5.1.

$$\Xi \mid \Gamma \vDash \mathit{handler2} \approx_{ctx} \mathit{handler1} : \mathbf{ServerConnT} \to 1$$

Our mechanized proof of the above theorem is done by showing logical relatedness in both directions and then appealing to the soundness of the logical relation (Theorem 7.3.2). Here we only discuss the proof of one direction:

$$\Xi \mid \Gamma \vDash \mathit{handler2} \leq_{\log} \mathit{handler1} : \mathbf{ServerConnT} \to 1$$

We use the rules for weakest preconditions and executions on the specification side explained above and make use of the relational specification given above for tables, which is justified by the CLWP-WP rule. A key element of the proof is the choice of predicate for relating the contents of the two tables. We use the following predicate:

$$\Phi_{\mathit{handlers}}(w, w') = \exists \mathit{sum} \in \mathbb{N}.\ w' = \mathit{sum} \wedge$$

$$\exists K.\ w = \texttt{cont}\left(K\left[\begin{array}{l}\texttt{let}\,(v, \mathit{reader}, \mathit{writer}) = - \,\texttt{in}\\ \\ \mathbf{loop}\ (\mathit{sum} + v)\ \mathit{reader}\ \mathit{writer}\end{array}\right]\right)$$

It essentially states that the values that are related in the two tables are: a captured continuation, on the implementation side, and a number, on the specification side. Furthermore, there is a number, *sum*, which is intuitively the sum so far. The natural number on the specification side is exactly this *sum*. The captured continuation on the implementation side is a continuation under some evaluation context K (existentially quantified). When resumed with a new value and connection, the stored continuation calls **loop** with the

new connection along with *sum* plus that value. The relation $\Phi_{handlers}$ above is indeed capturing the essence of the intuitive reason why the two implementations of handlers have contextually equivalent behavior.

According to the definition of our logical relations, to show logical relatedness we need to show that given any two related contexts the two programs behave in a related way. Since at the time of picking the predicate above we do not know what contexts we will have to operate under, we have to consider that our code of interest is inside some arbitrary (hence existentially quantified) evaluation context.

Note that the captured continuation mentioned in $\Phi_{handlers}$ refers to **loop**, which internally (see the code in Figure 7.1), uses the table itself. This is the reason why we need stronger relational specification for the table mentioned above.

7.6 Case study: one-shot call/cc

In this section we consider a more technical verification challenge involving continuations, due to Friedman and Haynes (1985). The challenge is to show that `call/cc` can be implemented using references and one-shot continuations, i.e., continuations that can only be called once. This problem has been studied for *sequential* higher-order languages with references in Dreyer, Neis, and Birkedal (2012) and Støvring and Lassen (2007), with pen-and-paper proofs, not mechanized formal verication. Here we show that the equivalence also holds in our concurrent language (subtly so; because we are using *may* contextual equivalence) and we give a mechanized formal proof thereof. Our proof is inspired by the proof of Dreyer, Neis, and Birkedal (2012), but we use a more involved invariant because of concurrency.

First, we define a polymorphic higher-order function that given a function f calls f with the current continuation:

$$\mathfrak{CC} \triangleq \Lambda\, \lambda f.\ \texttt{call/cc}\,(x.\ f\ x)$$

Note that $\mathfrak{CC}$ has type $\cdot \mid \cdot \vdash \mathfrak{CC} : \forall X.\, (\texttt{cont}(X) \to X) \to X$. Next, we will define a variant $\mathfrak{CC}'$ using one-shot continuations, and then prove the contextual equivalence of $\mathfrak{CC}$ and $\mathfrak{CC}'$.

To this end, we first define one-shot continuations $\mathfrak{CC}_1$ as follows:

$\mathfrak{CC}_1 \triangleq \Lambda\, \lambda f.\ \texttt{let}\, b = \texttt{false}\, \texttt{in}$

$$\texttt{call/cc}\,(x.\,f\ (\texttt{cont}(\texttt{let}\,y = -\,\texttt{in}\,\texttt{if}\,!b\,\texttt{then}\,\Omega\,\texttt{else}\,b \leftarrow \texttt{true}; \texttt{throw}\,y\,\texttt{to}\,x)))$$

Here Ω is the trivially diverging expression. Note that $\cdot \mid \cdot \vdash \mathfrak{CC}_1 : \forall X.\,(\texttt{cont}(X) \to X) \to X$. When applied, the one-shot continuation, $\mathfrak{CC}_1$, first allocates a *one-shot bit* b and then calls the given function with a continuation that uses b to ensure that the continuation is only called once.

Using one-shot continuations, we now define $\mathfrak{CC}'$:

$$\mathfrak{CC}' \triangleq \Lambda\,\lambda f.\ \texttt{let}\,\ell = \texttt{ref}(\texttt{cont}(-))\,\texttt{in}\,G\ f$$

$$G \triangleq \texttt{rec}\,G(f) =$$

$$\texttt{let}\,x =$$

$$\mathfrak{CC}_1\ \_\ (\lambda y.\,\ell \leftarrow y; f\ (\texttt{cont}(\texttt{throw}\,\texttt{cont}(-)\,\texttt{to}\,!\ell)))$$

$$\texttt{in}\,\mathfrak{CC}_1\ \_\ (\lambda y.\,G\ (\lambda\_.\,\texttt{throw}\,x\,\texttt{to}\,y))$$

The expression $\mathfrak{CC}'$ above has the same type as $\mathfrak{CC}$. $\mathfrak{CC}'$ perhaps looks fairly complex but the intuition is straightforward. It first allocates ℓ with the trivial continuation, then it takes a one-shot continuation and updates ℓ. When the one-shot continuation is used, it will first grab another *fresh* one-shot continuation and update ℓ with it before continuing. Hence, intuitively, every time the one-shot continuation stored in ℓ is used, it is immediately refreshed with an unused one, thus mimicking the behavior of $\mathfrak{CC}$.

We now prove that $\mathfrak{CC}$ is contextually equivalent to $\mathfrak{CC}'$:

Theorem 7.6.1.

$$\cdot \mid \cdot \vDash \mathfrak{CC} \approx_{ctx} \mathfrak{CC}' : \forall X.\,(\texttt{cont}(X) \to X) \to X$$

We only discuss one side of the refinement, namely, $\mathfrak{CC}' \leq_{\text{ctx}} \mathfrak{CC}$. The proof of the other side is similar but simpler.

Our proof is similar to the one by Dreyer, Neis, and Birkedal (2012), except for the invariant that is used to prove relatedness.[4] Translated to our setting, the

[4] In the work of Dreyer, Neis, and Birkedal (2012), invariants were called *islands*.

invariant used by Dreyer, Neis, and Birkedal (2012) is:

$$\boxed{\begin{array}{l}\exists b.\ b \mapsto_i \texttt{false} * \\ \\ \ell \mapsto_i \texttt{cont}\left(\begin{array}{l}\texttt{let}\, y = - \,\texttt{in}\, \texttt{if}\, !\, b\, \texttt{then}\, \Omega\, \texttt{else} \\ \\ b \leftarrow \texttt{true}; \texttt{throw}\, y\, \texttt{to}\, \texttt{cont}(K[\mathit{restore}(\ell)])\end{array}\right)\end{array}}^{\mathcal{N}.\mathfrak{CC}}$$

where

$$\mathit{restore}(\ell) \triangleq \texttt{let}\, x = - \,\texttt{in}\, \mathfrak{CC}_1\ \_\ (\lambda y.\, G\ (\lambda \_.\, \texttt{throw}\, x\, \texttt{to}\, y))$$

Here K is the continuation that is captured by $\mathfrak{CC}$. Intuitively, it states that the continuation stored in ℓ is a one-shot continuation that has never been used (as the one-shot bit b stores `false`). Furthermore, whenever used it will first restore ℓ with a fresh one-shot continuation (using the nested evaluation context $\mathit{restore}(\ell)$).

This invariant suffices for a *sequential* programming language. However, in our concurrent settting, the "continuation" captured by $\mathfrak{CC}'$ may be shared among multiple threads and, if they use it concurrently, a race may occur. In other words, it may happen that a thread is using the continuation captured by $\mathfrak{CC}'$ and before this thread manages to capture another one-shot continuation and restore ℓ, another thread attempts to use the then invalid one-shot continuation, and hence it diverges.

We prove that the contextual refinement still holds (despite the possibility of divergence). However, because of the possible racing, we need to use a weaker invariant:

$$\boxed{\begin{array}{l}\exists b, M.\ \mathit{OneShotBits}(M) * \mathit{isOneShotBit}(b) * \\ \\ \left(\ast_{r \in M}\ \exists v \in \{\texttt{true}, \texttt{false}\}\,.\ r \mapsto_i v\right) * \\ \\ \ell \mapsto_i \texttt{cont}\left(\begin{array}{l}\texttt{let}\, y = - \,\texttt{in}\, \texttt{if}\, !\, b\, \texttt{then}\, \Omega\, \texttt{else} \\ \\ b \leftarrow \texttt{true}; \texttt{throw}\, y\, \texttt{to}\, \texttt{cont}(K[\mathit{restore}(\ell)])\end{array}\right)\end{array}}^{\mathcal{N}.\mathfrak{CC}}$$

This invariant says that ℓ stores a one-shot continuation with a one-shot bit b and that we have a set of bits that, intuitively, have been associated to one-shot continuations. We also know that b is one such one-shot bit, $\mathit{isOneShotBit}(b)$. The predicates $\mathit{OneShotBits}()$ and $\mathit{isOneShotBit}()$ are defined using iris resources. Details can be found in Appendix F. Here we only need to

know two things about them, namely that $\mathit{isOneShotBit}(b)$ is persistent and that

$$\mathit{OneShotBits}(M) * \mathit{isOneShotBit}(b) \vdash b \in M \qquad \text{(in-bits)}$$

Persistence allows us to retain the information $\mathit{isOneShotBit}(b)$ once we have opened the invariant and have read ℓ. Due to the race condition explained above, when we open the invariant we know, by (in-bits), that there is a value $v \in \{\texttt{true}, \texttt{false}\}$ stored in b, and this suffices for being able to complete the refinement proof.

The other refinement, $\mathfrak{CC} \leq_{\text{ctx}} \mathfrak{CC}'$ is simpler and follows basically using the same argument as in Dreyer, Neis, and Birkedal (2012). This makes sense intuitively because we simply have to show that *there exists* an execution on the specification side that converges.

7.7 Mechanization in Coq

Taking advantage of the Coq formalization of Iris and Iris Proof Mode (IPM) (Krebbers, Timany, and Birkedal, 2017), we have mechanized all the technical development and results in Coq. This includes mechanizing the small-step operational semantics of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ and instantiating Iris with it. Our Coq development is about 9900 lines and includes proofs of contextual refinements for pairs of fine-grained/coarse-grained stacks and counters which we omitted discussion of for reasons of space.

For binders, we use the Autosubst library (Schäfer, Tebbi, and Smolka, 2015) which facilitates the use of de Bruijn indices by providing support for simplification of substitutions. In $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, evaluation contexts are also values and hence also expressions. This forces us to define these mutually inductively. This means that we need to derive the induction principle for these inductive types in Coq by hand. Furthermore, we have to help Autosubst in deriving substitution and simplification lemmas for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ that it should otherwise automatically infer. This is mainly why the definition of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ itself takes up about 10% of the whole Coq development.

7.8 Related work

There has been a considerable body of work on (delimited) continuations, but, we are not aware of any logics or relational models for reasoning about concurrent programs with continuations, let alone a mechanized framework for relational verification of concurrent programs with continuations.

Program logics for reasoning about continuations Delbianco and Nanevski (2013) present a type theory for Hoare-style reasoning about an imperative higher-order programming language with (algebraic) continuations, but without concurrency. The system of Delbianco and Nanevski (2013) does not allow higher-order code (including continuations) to be stored in the heap. Note that storing higher-order code in the heap is essential for both implementing the continuation-based web servers and implementing continuations in terms of one-shot continuations. Crolard and Polonowski (2012) develop a program logic for reasoning about jumps but their sequential programming language features no heap or recursive types. Berger (2010) presents a program logic for reasoning about programs in a programming language which is essentially an extension of PCF (Plotkin, 1977) with continuations.

Relational reasoning about continuations The work most closely related to ours is that of Dreyer, Neis, and Birkedal (2012) who consider a variety of different stateful programming languages and investigate the impact of the higher-order state and control effects (including `call/cc` and `throw`). In contrast to our work, they do not consider concurrency. Moreover, they reason directly in a model, whereas we define our logical relation using a program logic (Iris), which means that we can reason more compositionally and at a higher level of abstraction. Another advantage of using Iris, is that we have been able to leverage its Coq formalization and thus to mechanize all of our development. As mentioned in Section 7.6, our proof that continuations can be expressed in terms of one-shot continuations is inspired by loc. cit.

There are several other works on relational reasoning for sequential programming languages with continuations, e.g., Felleisen and Hieb (1992), Laird (1997), and Støvring and Lassen (2007). These differ from our work at least in that they do not consider concurrency.

Relational reasoning about concurrency There has been much work on relational reasoning about concurrent higher-order imperative programs, without continuations. The work most closely related to ours also is that of Krebbers,

Timany, and Birkedal (2017), who develop mechanized logical relations (in Iris) for reasoning about contextual equivalence of programs in $\mathsf{F}_{\mu,ref,conc}$, a language similar to the one we consider but without `call/cc` and `throw`. The approach in *loc. cit.* is based on earlier, non-mechanized logical relations for fine-grained concurrent programs (Birkedal, Sieczkowski, and Thamsborg, 2012; Turon, Dreyer, and Birkedal, 2013; Turon, Thamsborg, Ahmed, Birkedal, and Dreyer, 2013). These relational models give an alternative method to linearizability (Herlihy and Wing, 1990) for reasoning about contextual refinement for fine-grained concurrent programs. The logical relations method also works in the presence of higher-order programs, which linearizability traditionally struggles with, although there has been some recent promising developments (Cerone, Gotsman, and Yang, 2014; Murawski and Tzevelekos, 2017). In this paper, we have extended the method of logical relations for reasoning about contextual refinement for higher-order fine-grained concurrent programs to work for programs that also use continuations.

7.9 Conclusion and future work

We have developed a logical relation for $\mathsf{F}^{\mu,ref}_{conc,cc}$, a programming language with advanced features such as impredicative polymorphism à la system F, higher-order mutable references, recursive types, concurrency and most notably continuations. We have devised new non-context-local proof rules for reasoning about weakest preconditions in Iris in the presence of continuations and also introduced context-local weakest preconditions for regaining context-local reasoning about expressions that do not involve non-local control flow. We have defined our relational model and proved properties thereof in the Iris program logic framework. This has greatly simplified the definition of our relational model, the existence of which is non-trivial because of the type-world circularity (Ahmed, Appel, and Virga, 2002; Ahmed, 2004; Birkedal, Reus, Schwinghammer, Støvring, Thamsborg, and Yang, 2011). Furthermore, working inside Iris has enabled us to mechanize the entire development presented in this paper on top of the Coq proof assistant.

We have demonstrated how our logical relation can be used to establish contextual equivalence for a pair of simplified web-server implementations: one storing the state explicitly and one storing the current continuation. The application of context local reasoning in the middle of our logical relatedness proofs demonstrates the usefulness and versatility of context-local weakest preconditions. Finally, we have also given the first (mechanized) proof of the correctness of Friedman and Haynes (1985) encoding of continuations by means of one-shot continuations in a concurrent programming language.

In the future, we wish to extend our mechanization to reason about delimited continuations (Danvy and Filinski, 1990; Felleisen, 1988). Currently our mechanized reasoning is done interactively, in the same style as one reasons in Coq. In the future, we would also like to complement that with more automated reasoning methods.

Chapter 8

Conclusion and future work

Correctness of safety- and security-critical software is crucial. An important tool in achieving high assurance of safety and security of such software systems is type theory. Compilers of statically typed programming languages like OCaml, Haskell, etc. use types to make sure that programs are well-behaved, i.e., they will not crash, before they compile programs. On the other hand, dependent type theories form the basis of proof assistants like the calculus of constructions (CoC) and its extensions forming the basis of the proof assistant Coq. Proof assistants are important tools for mathematicians and computer scientists that allow us to formally verify the correctness of a mathematical theories. This ranges from mathematical developments like category theory to the study of programming languages and proving correctness and safety of programs. In the course of this thesis we presented contributions to both of these applications of types. We used and contributed to the type theory of the Coq proof assistant. We also studied programs and programming languages through their types.

In Chapter 3 we presented a formal development of category theory in the proof assistant Coq. This development encompasses most of the basic category theoretical constructions, i.e., not including higher or enriched categories. As such, we believe that this development is fit to be used as a library for formalization of theories that build on categories, e.g., categorical semantics of type theory and logic. In particular, an interesting future work is to develop a general framework for working with categorical logic. That is, an extensible language of connectives of predicate logic that is interpreted in general on top of a suitable family of categories that have enough structure to interpret predicate logic, e.g., (sheaf/presheaf) toposes. This would allow us to easily add more connectives and modalities to the logic, e.g., the later modality in the topos

of trees (Birkedal, Mogelberg, Schwinghammer, and Stovring, 2011). Such a setup would be best complemented with an interactive proof mode similar to the one that is presented for the Iris program logic (Krebbers, Timany, and Birkedal, 2017). Such a setup would allow us to easily study different logics and their extensions both externally (working with the objects and morphism of the category itself) and internally (working with the logic of the category through the interactive proof mode).

The development of category theory presented in Chapter 3 uses Coq's universes to formalize the concept of relative smallness and largeness. This feature of the development points out a limitation in the predicative calculus of inductive constructions (pCIC), the underlying logic of the proof assistant Coq at time of that development. This limitation, i.e., lack of the cumulativity (subtyping) relation for inductive types in pCIC has since been alleviated by the introduction of the predicative calculus of cumulative inductive constructions (pCuIC) which is presented in Chapter 4. The type system of pCuIC extends that of pCIC with rules for cumulativity and conversion of inductive types. The cumulativity relation of pCuIC for categories corresponds precisely to smallness and largeness of categories as expected, i.e., any small category is also a large category. This extension also applies to inductive types that do not involve the mathematical notion of smallness and largeness. For instance, due to cumulativity, for a type $A : \texttt{Type}_i$ we have $A : \texttt{Type}_j$ for any $i \leq j$. So, we can construct types $list_i\ A$ and $list_j\ A$, where $list_k$ is the universe-polymorphic inductive type of lists instantiated with universe $\texttt{Type}_k$. In pCuIC inductive types $list_i\ A$ and $list_j\ A$ are considered subtypes of one another and also *judgementally equal*.

Chapter 4 presents a proof of consistency of pCuIC by constructing a model in ZFC with the added axiom that there is a countable hierarchy of strictly increasing uncountable strongly inaccessible cardinals. This model is based on and inspired by the model of pCIC constructed by Lee and Werner (2011). We have used the axiom of choice in this construction to prove that the fixpoint that is taken to construct the model the inductive types does indeed belong to the *set-theoretic* universe that it should, i.e., the set-theoretic universe corresponding to the type-theoretic universe that is its arity. The use of the axiom of choice does not seem absolutely necessary. We only need it to show that the closure ordinal for the fixpoint is in the necessary universe, a fact that can be easily shown to be equivalent to the fixpoint itself being in the universe. Further research is required to determine whether or not the axiom of choice (or potentially a weaker axiom) is indeed necessary for proving that fixpoints of inductive types are in the expected universe. For instance, hypothetically one could argue based on the syntactic criteria for well-formedness of inductive types, that for the equations that arise from inductive types the closing ordinal, or the fixpoint itself, does belong to the required set-theoretic universe. We have not managed

to find such a proof. Our model, like the model of Lee and Werner (2011), does not support inductive types in the sort `Prop`. Future research is necessary to shed light on how to incorporate inductive types in `Prop` into our model. This problem is related to the previous one and is in fact made more complicated by it. In particular, the way that we have reasoned about fixpoints of inductive types being in the required universe relies on the fact that all of the inductive types in a mutual inductive block are in the same arity. Considering inductive types in `Prop` breaks this assumption: an inductive type I in $\texttt{Type}_i$ can be an argument of a constructor of an inductive type J in `Prop`, and J can be an argument of a constructor of an inductive type K in $\texttt{Type}_k$. Notice, that this does not impose the restriction $i \leq k$ as it would, had I been an argument of a constructor of K. One other restriction of our system with respect to Coq is the requirement that parameters of inductive types must be uniform, i.e., fixed for the entire mutual inductive block. The only part of our construction that needs this assumption is the proof that the interpretation of inductive types are in the set-theoretic universe corresponding to their arity (Lemma B.3.8). Future work on set-theoretic models of Coq includes constructing a set-theoretic model that allows for inductive types with heterogeneous parameters. Finally, we believe that the model of Barras (2012) for pCIC which he uses to prove strong normalization can be adapted to support the extensions that pCuIC offers similarly to how we have adapted the model of Lee and Werner (2011). Note that strong normalization is strictly stronger than (implies) consistency of the type system that we have proven in this thesis.

Chapter 5 presented the development of unary and binary operationally-based logical relations models for $\mathsf{F}_{\mu,ref,conc}$, respectively to prove type soundness and to establish equivalence of programs. Operationally-based logical relations have the benefit that they are modular, i.e., separate parts of the code can be proven separately (possibly through different approaches) and proofs can be composed. Furthermore, as a semantics-based approach they guarantee that the abstraction barriers within programs are respected. Moreover, for programming languages featuring advanced types, e.g., a combination of polymorphism and dynamically allocated higher-order references, operationally-based logical relations are much easier to develop and use compared to approaches based on denotational semantics. The logical relations models that we presented in this chapter and those that follow it are developed on top of the Iris program logic. The benefits of developing these models in Iris are twofold. First and most importantly the logic of Iris comes equipped with advanced reasoning principles that allow a direct inductive encoding of our logical relations models. That is, these features allow us reason about intricate details of the model, e.g., step-indices, at a high level of abstraction. The second benefit of working in Iris is the formalization of the Iris program logic on top of the Coq proof assistant which we use as a Coq library to formalize our logical relations models

and the results obtained from them in Coq. The development presented in Chapter 5 is the first formalization of logical relations, in a proof assistant, for a programming language with a type system as rich as the type system of $\mathsf{F}_{\mu,ref,conc}$.

The approach taken in Chapter 5, i.e., developing logical relations models in the Iris program logic, is quite general and versatile. This methodology can be applied to other programming languages and problems as we have done in Chapters 6 and 7 of this very thesis. Future research in this area includes applying this methodology to other programming language features and problems. In particular, the study of compilers would benefit from this methodology. Logical relations models can be used to prove correctness (Benton and Hur, 2009) and security of compilers (Devriese, Patrignani, and Piessens, 2016).

In Chapter 6 we studied proper encapsulation of state by a Haskell-style ST monad. We did this by studying equivalence of programs in the presence of the ST monad in the programming language STLang. This was an open problem for about two decades. Future research is required to show that monadic encapsulation of state by the ST monad does indeed encapsulate state properly in the presence of other programming language features, e.g., concurrency. This is specially interesting as it is not immediately clear how one can extend IC predicates that we have used for defining our logical relations model to a language with concurrency. There is another important problem regarding monadic encapsulation of state that deserves attention: "is STLang indeed pure?" But to answer that question one needs to give a formal definition of purity in the presence of state. These problems are important and quite interesting. However, they require more exploratory research and it is not immediately clear how to begin to attack these problems.

Chapter 7 studied a programming language $\mathsf{F}^{\mu,ref}_{conc,cc}$ which is an extension of $\mathsf{F}_{\mu,ref,conc}$ with first-class continuations. There we presented unary and binary logical relations models for this programming language. We used these logical relations models to prove type safety and to prove equivalence of programs. Continuations allow programs to be suspended and later resumed. As a result programs cannot be verified in isolation as the context under which they are running can be captured into a continuation. Technically, in Iris this means that weakest precondition proposition predicates that we use for program verification are not *context-local* anymore. Hence, we presented so called context-local weakest preconditions. This variant of weakest precondition predicates not only guarantee the safety of the program but it also requires the program to behave (according to the operational semantics) as though it does not involve continuations. Hence, context-local weakest preconditions can be used to verify parts of programs that do not involve continuations in a disruptive way. We used

context-local weakest preconditions together with our binary logical relations model to establish equivalence of two web server implementations where one uses continuations to store the state of its communication with the client and the other stores the state directly.

Arguably one of the most important use cases of continuations in programming languages is to mimic concurrency. There are many libraries for different programming languages that provide this functionality, usually referred to as *green threads* or *light-weight threads*. Creation and using of green threads is much more efficient compared to genuine threads provided by the operating system; hence the name. For this reason they are quite useful and are in fact often used in conjunction with real concurrency to run multiple green threads at once. The development presented in Chapter 7 provides a suitable platform for proving correctness of green threads and more interestingly to show that programs using green threads are equivalent to programs using actual concurrency and that hence programmers can switch between the two paradigms or a combination thereof without any change to the overall behavior of programs.

Appendix A

The formal definition of pCIC

Notations for different equalities and the like:

$a \triangleq b$	a is defined as b
$a = b$	equality of mathematical objects, e.g., sets, sequences, etc.
$t \equiv t'$	Syntactically identical terms
$\Gamma \vdash t \simeq t' : A$	Judgemental equality, i.e., t and t' are judgementally equal terms of type A under context Γ

A.1 Syntax of pCIC

In this section we present the predicative calculus of inductive constructions (pCIC). The terms and contexts of the language pCIC is as follows:

$$x, y, z, \ldots \qquad \text{(Variables)}$$

$$s ::= \texttt{Prop}, \texttt{Type}_0, \texttt{Type}_1, \ldots \qquad \text{(Sorts)}$$

$$t, u, \ldots,$$

$$M, N, \ldots,$$

$$A, B, \ldots ::= x \mid s \mid \mathbf{\Pi} x : A.\, B \mid \lambda x : A.\, t \mid \qquad \text{(Terms)}$$

$$\mathbf{let}\, x := t : A\, \mathbf{in}\, u \mid M\ N \mid$$

$$\mathcal{D}.x \mid$$

$$\mathbf{Elim}(t; \mathcal{D}.d_i; \vec{u}; Q_{d_1}, \ldots, Q_{d_n})\,\{f_{c_1}, \ldots, f_{c_m}\}$$

$$\mathcal{D} ::= \mathbf{Ind}_n\,\{\Delta_I := \Delta_C\} \qquad \text{(Inductive blocks)}$$

$$\Delta ::= \cdot \mid \Delta, x : A \qquad \text{(Declarations)}$$

$$\Gamma ::= \cdot \mid \Gamma, x : A, \mid \Gamma, x := t : A \mid \Gamma, \mathcal{D} \qquad \text{(Contexts)}$$

Note that although by abuse of notation we write $x : A \in \Gamma$, $x := t : A \in \Gamma$ or $x : A \in \Delta$, contexts and declarations *are not* sets. In particular, the order in declarations is important, this is even the case for declarations where there can be no dependence among the elements. We write $\Delta(x)$ to refer to A whenever $x : A \in \Delta$.

Here, $\mathbf{Ind}_n\,\{\Delta_I := \Delta_C\}$ is a block of mutual inductive definitions where n is the number of parameters, the declarations in Δ_I are the inductive types of the block and declarations in Δ_C are constructors of the block. The term $\mathbf{Ind}_n\,\{\Delta_I := \Delta_C\}.x$ is an inductive definition whenever $x \in \Delta_I$ and a constructor whenever $x \in \Delta_C$. The term $\mathbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_n})\,\{f_{c_1}, \ldots, f_{c_m}\}$ is the elimination of t (a term of the inductive type $\mathcal{D}.d_k$ applied to parameters $\vec{u}$), Q_{d_i}'s are the motives of elimination, i.e., the result of the elimination will have type $Q_{d_k}\ \vec{u}\ \vec{a}\ t$ whenever the term being eliminated, t, has type $\mathcal{D}.d_k\ \vec{u}\ \vec{a}$. The term f_{c_i} in the eliminator above is a *case-eliminator* corresponding to the case where t is constructed using constructor $\mathcal{D}.c_i$.

We write $\mathsf{len}(\vec{s})$, $\mathsf{len}(p)$, $\mathsf{dom}(f)$, $\mathsf{dom}(\Delta)$ and $\mathsf{dom}(\Gamma)$ respectively for the length of a sequence $\vec{s}$, length of a tuple p, the domain of a partial map f, domain of a declaration Δ or domain of a context Γ. Notice crucially that the inductive types and constructors of an inductive block are *not* part of the domain of the context that they appear in. We write nil for the empty sequence. We write $\mathit{Inds}(\Gamma)$ for the sequence of inductive types in the context Γ. We write tuples as $\langle a_1; a_2; \ldots; a_n \rangle$.

Definition A.1.1 (Free variables)**.**

Free variables of terms

$$FV(\texttt{Prop}) \triangleq \emptyset$$

$$FV(\texttt{Type}_i) \triangleq \emptyset$$

$$FV(z) \triangleq \{z\}$$

$$FV(\lambda y : A.\, u) \triangleq FV(A) \cup (FV(u) \setminus \{y\})$$

$$FV(\mathbf{let}\, y := u : A\, \mathbf{in}\, v) \triangleq FV(A) \cup FV(u) \cup (FV(v) \setminus \{y\})$$

$$FV(u\ v) \triangleq FV(u) \cup FV(v)$$

$$FV(\mathcal{D}.z) \triangleq FV(\mathcal{D})$$

$$FV(\mathbf{Elim}(u; \mathcal{D}.d_i; \vec{u}; \vec{Q})\left\{\vec{f}\right\}) \triangleq FV(u) \cup FV(\mathcal{D}) \cup FV(\vec{u}) \cup FV(\vec{Q}) \cup FV(\vec{f})$$

Free variables of inductive blocks

$$FV(\mathbf{Ind}_n\{\Delta_I := \Delta_C\}) \triangleq FV(\Delta_I) \cup FV(\Delta_C)$$

Free variables of sequences of terms

$$FV(\mathsf{nil}) \triangleq \mathsf{nil}$$

$$FV(v, \vec{v}) \triangleq FV(v) \cup FV(\vec{v})$$

Free variables of declarations

$$FV(\cdot) \triangleq \emptyset$$

$$FV(y : A, \Delta) \triangleq FV(A) \cup FV(\Delta)$$

We define simultaneous substitution for terms as follows:

Definition A.1.2 (Simultaneous substitution)**.** *We assume that $\vec{x}$ is a sequence of distinct variables. In this definition for the sake of simplicity we use y for all bound variables.*

Substitution for terms

$$\mathtt{Prop}[\vec{t}/\vec{x}] \triangleq \mathtt{Prop}$$

$$\mathtt{Type}_i[\vec{t}/\vec{x}] \triangleq \mathtt{Type}_i$$

$$z[\vec{t}/\vec{x}] \triangleq t_i \ \textit{if}\ x_i = z$$

$$z[\vec{t}/\vec{x}] \triangleq z \ \textit{if}\ \forall i.\ x_i \neq z$$

$$(\lambda y : A.\, u)[\vec{t}/\vec{x}] \triangleq \lambda y : A[\vec{t}/\vec{x}].\, u[\vec{t}'/\vec{x}']$$

$$(\mathbf{let}\, y := u : A\, \mathbf{in}\, v)[\vec{t}/\vec{x}] \triangleq \mathbf{let}\, y := u[\vec{t}/\vec{x}] : A[\vec{t}/\vec{x}]\, \mathbf{in}\, v[\vec{t}'/\vec{x}']$$

$$(u\ v)[\vec{t}/\vec{x}] \triangleq u[\vec{t}/\vec{x}]\ v[\vec{t}/\vec{x}]$$

$$\mathsf{Ind}_n\left\{\Delta_I := \Delta_C\right\}.z[\vec{t}/\vec{x}] \triangleq \mathsf{Ind}_n\left\{\Delta_I[\vec{t}/\vec{x}] := \Delta_C[\vec{t}/\vec{x}]\right\}.z$$

$$\mathsf{Elim}(u; \mathcal{D}.d_i; \vec{u}; \vec{Q})\left\{\vec{f}\right\}[\vec{t}/\vec{x}] \triangleq \mathsf{Elim}(u[\vec{t}/\vec{x}]; \mathcal{D}.d_i[\vec{t}/\vec{x}]; \vec{u}[\vec{t}/\vec{x}])$$
$$\left\{\vec{Q}[\vec{t}/\vec{x}]\right\}\vec{f}[\vec{t}/\vec{x}]$$

Substitution for inductive blocks

$$\mathsf{Ind}_n\left\{\Delta_I := \Delta_C\right\}[\vec{t}/\vec{x}] \triangleq \mathsf{Ind}_n\left\{\Delta_I[\vec{t}/\vec{x}] := \Delta_C[\vec{t}/\vec{x}]\right\}$$

Substitution for sequences

$$\mathsf{nil}[\vec{t}/\vec{x}] \triangleq \mathsf{nil}$$

$$v, \vec{v}[\vec{t}/\vec{x}] \triangleq v[\vec{t}/\vec{x}], \vec{v}[\vec{t}/\vec{x}]$$

Substitution for declarations

$$\cdot[\vec{t}/\vec{x}] \triangleq \cdot$$

$$y : A, \Delta[\vec{t}/\vec{x}] \triangleq y : A[\vec{t}/\vec{x}], \Delta[\vec{t}/\vec{x}]$$

Substitution for context

$$\cdot[\vec{t}/\vec{x}] \triangleq \cdot$$

$$y : A, \Gamma[\vec{t}/\vec{x}] \triangleq y : A[\vec{t}/\vec{x}], \Gamma[\vec{t'}/\vec{x'}]$$

where $\vec{x'} = \vec{x}$ *and* $\vec{t'} = \vec{t}$ *if* y *does not appear in* $\vec{x}$ *and is as follows whenever* $x_i = y$*:*

$$\vec{x'} = x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$$
$$\vec{t'} = t_0, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$$

A.2 Basic constructions

Figure A.1 shows typing rules for the basic constructions, i.e., well-formedness of contexts ($\mathcal{WF}(\Gamma)$), sorts, let bindings, dependent products (also referred to as dependent functions), lambda abstractions and applications. It also contains the rules for the judgemental equality for these constructions. In this figure, the relation $\mathcal{R}_s$ indicates the sort that a (dependent) product type belongs to. The sort `Prop` is *impredicative* and therefore any product type with codomain in `Prop` also belongs to `Prop`.

$$\textsc{WF-ctx-empty}\quad \mathcal{WF}(\cdot)$$

$$\textsc{WF-ctx-hyp}\quad \frac{\Gamma \vdash A : s \qquad x \notin \mathsf{dom}(\Gamma)}{\mathcal{WF}(\Gamma, x : A)}$$

$$\textsc{WF-ctx-def}\quad \frac{\Gamma \vdash t : A \qquad x \notin \mathsf{dom}(\Gamma)}{\mathcal{WF}(\Gamma, (x := t : A))}$$

$$\textsc{Prop}\quad \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathtt{Prop} : \mathtt{Type}_i}$$

$$\textsc{Hierarchy}\quad \frac{\mathcal{WF}(\Gamma) \qquad i < j}{\Gamma \vdash \mathtt{Type}_i : \mathtt{Type}_j}$$

$$\textsc{Let}\quad \frac{\Gamma, (x := t : A) \vdash u : B}{\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u : B[t/x]}$$

$$\textsc{Var}\quad \frac{\mathcal{WF}(\Gamma) \qquad x : A \in \Gamma \qquad \text{or} \qquad (x := t : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Let-eq}\quad \frac{\Gamma \vdash A \simeq A' : s \qquad \Gamma \vdash t \simeq t' : A \qquad \Gamma, (x := t : A) \vdash u \simeq u' : B}{\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u \simeq \mathbf{let}\, x := t' : A'\, \mathbf{in}\, u' : B[t/x]}$$

$$\textsc{Prod}\quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \mathbf{\Pi} x : A.\, B : s_3}$$

$$\textsc{Prod-eq}\quad \frac{\Gamma \vdash A \simeq A' : s_1 \qquad \Gamma, x : A \vdash B \simeq B' : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \mathbf{\Pi} x : A.\, B \simeq \mathbf{\Pi} x : A'.\, B' : s_3}$$

$$\textsc{Lam}\quad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \mathbf{\Pi} x : A.\, B : s}{\Gamma \vdash \lambda x : A.\, M : \mathbf{\Pi} x : A.\, B}$$

$$\textsc{App}\quad \frac{\Gamma \vdash M : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\textsc{Lam-eq}\quad \frac{\Gamma \vdash A \simeq A' : s_1 \qquad \Gamma, x : A \vdash M \simeq M' : B \qquad \Gamma \vdash \mathbf{\Pi} x : A.\, B : s_2}{\Gamma \vdash \lambda x : A.\, M \simeq \lambda x : A'.\, M' : \mathbf{\Pi} x : A.\, B}$$

$$\textsc{App-eq}\quad \frac{\Gamma \vdash M \simeq M' : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N \simeq N' : A}{\Gamma \vdash M\ N \simeq M'\ N' : B[N/x]}$$

$$\textsc{Predicativity}\quad \mathcal{R}_s(\mathtt{Type}_i, \mathtt{Type}_j, \mathtt{Type}_{\mathsf{max}\{i,j\}})$$

$$\textsc{Predicativity'}\quad \mathcal{R}_s(\mathtt{Prop}, \mathtt{Type}_i, \mathtt{Type}_i)$$

$$\textsc{Impredicativity}\quad \mathcal{R}_s(s, \mathtt{Prop}, \mathtt{Prop})$$

Figure A.1: Basic construction

A.3 Inductive types and their eliminators

We consider blocks of predicative (not in `Prop`) mutual inductive types. Most inductive types in `Prop` can be encoded using their Church encoding. The typing rules for inductive types, their constructors and eliminators are depicted in Figure A.2.

Well-formedness of inductive types The first rule in this figure is the well-formedness of inductive types. It states that in order to have that the context Γ is well-formed after adding the mutual inductive block $\mathbf{Ind}_n\{\Delta_I := \Delta_C\}$, i.e., $\mathcal{WF}(\Gamma, \mathbf{Ind}_n\{\Delta_I := \Delta_C\})$, all inductive types in the block as well as all constructors need to be well-typed terms. In addition, we must have that the well-formedness side condition $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ holds. The well-formedness side condition $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ holds whenever:

- All variables in Δ_I and Δ_C are distinct.
- The first n arguments of all inductive types and constructors in the block are the parameters. In other words, there is a sequence of terms $\vec{P}$ such that $\mathsf{len}(\vec{P}) = n$ and for all $x : T \in \Delta_I, \Delta_C$ we have $T \equiv \mathbf{\Pi}\vec{p} : \vec{P}.\, U$ for some U.
- Parameters are parametric. In other words, for all $c : T \in \Delta_C$ we have $T \equiv \mathbf{\Pi}\vec{p} : \vec{P}.\, \mathbf{\Pi}\vec{x} : \vec{B}.\, d\ \vec{p}\ \vec{v}$. That is, every constructor constructs a term of some inductive type in the block where values applied for parameter arguments of the inductive type are parameter arguments of the constructor. Parameters must be fixed in the whole inductive block. That is, we do not support heterogeneous parameters where an inductive type of the block can be an argument of a constructor with different parameters than the inductive family being defined.
- Every inductive type is just a type (an element of a universe) that depends on a number of arguments beginning with parameters. The non-parameter arguments are called the arity of the inductive type. In other words, for all $d \in \mathsf{dom}(\Delta_I)$ we have $\Delta_I(d) \equiv \mathbf{\Pi}\vec{x} : \vec{P}.\, \mathbf{\Pi}\vec{m} : \vec{M}.\, A_d$ where $\vec{M}$ are called the indices of the inductive type and A_d is a sort called the arity of the inductive type d. We require that $A_d \neq$ `Prop`.
- Every constructor is a constructs terms of an inductive type in the block. In other words, for all $c \in \mathsf{dom}(\Delta_C)$ we have $c \in \mathsf{Constrs}(\Delta_C, d)$ for some $d \in \Delta_I$ where $\mathsf{Constrs}(\Delta_C, d)$ is the set of constructors in Δ_C that construct terms of the inductive type d.

$$\mathsf{Constrs}(\Delta_C, d) \triangleq \left\{ c \in \mathsf{dom}(\Delta_C) \mid \Delta_C(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P}.\, \mathbf{\Pi}\vec{x} : \vec{U}.\, d\ \vec{u} \right\}$$

IND-WF

$$\frac{\begin{array}{c}\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) \\ (A \equiv \mathbf{\Pi} p : P.\, \mathbf{\Pi} m : M.\, A_d \qquad \Gamma \vdash A : s_d \text{ for all } (d : A) \in \Delta_I) \\ (T \equiv \mathbf{\Pi} p : P.\, T' \\ \Gamma, \Delta_I, p : P \vdash T' : A_d \text{ for all } (c : T) \in \Delta_C \text{ if } c \in \mathsf{Constrs}(\Delta_C, d))\end{array}}{\mathcal{WF}(\Gamma, \mathbf{Ind}_n \{\Delta_I := \Delta_C\})}$$

IND-TYPE

$$\frac{\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \qquad d_i \in \mathsf{dom}(\Delta_I)}{\Gamma \vdash \mathcal{D}.d_i : \Delta_I(d_i)}$$

IND-CONSTR

$$\frac{\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \qquad c \in \mathsf{dom}(\Delta_C)}{\Gamma \vdash \mathcal{D}.c : \Delta_C(c)[\overrightarrow{\Delta_I.d}/\vec{d}]}$$

IND-ELIM

$$\frac{\begin{array}{c}\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \\ \mathrm{dom}(\Delta_I) = \{d_1, \ldots, d_l\} \qquad \mathrm{dom}(\Delta_C) = \{c_1, \ldots, c_{l'}\} \\ (\Gamma \vdash Q_{d_i} : \mathbf{\Pi} \vec{x} : \vec{A}.\, (d_i\ \vec{x}) \to s' \text{ where} \\ \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{x} : \vec{A}.\, s \text{ for all } 1 \le i \le l) \\ \Gamma \vdash t : \mathcal{D}.d_k\ \vec{u}\ \vec{m} \qquad \mathsf{len}(\vec{u}) = n \\ \Gamma \vdash f_{c_i} : \xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i)) \text{ for all } 1 \le i \le l'\end{array}}{\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l}) \left\{f_{c_1}, \ldots, f_{c_{l'}}\right\} : Q_{d_k}\ \vec{u}\ \vec{m}\ t}$$

IND-ELIM-EQ

$$\frac{\begin{array}{c}\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \\ \mathrm{dom}(\Delta_I) = \{d_1, \ldots, d_l\} \qquad \mathrm{dom}(\Delta_C) = \{c_1, \ldots, c_{l'}\} \\ (\Gamma \vdash Q_{d_i} \simeq Q'_{d_i} : \mathbf{\Pi} \vec{p} : \vec{P}.\, \mathbf{\Pi} \vec{x} : \vec{A}.\, (d_i\ \vec{x}) \to s' \text{ where} \\ \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{p} : \vec{P}.\, \mathbf{\Pi} \vec{x} : \vec{A}.\, s \text{ for all } 1 \le i \le l) \\ \Gamma \vdash t \simeq t' : \mathcal{D}.d_k\ \vec{u}\ \vec{m} \qquad \mathsf{len}(\vec{u}) = n \qquad \Gamma \vdash \vec{u} \simeq \vec{u'} : \vec{P} \\ \Gamma \vdash f_{c_i} \simeq f'_{c_i} : \xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i)) \text{ for all } 1 \le i \le l'\end{array}}{\begin{array}{c}\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l}) \left\{f_{c_1}, \ldots, f_{c_{l'}}\right\} \simeq \\ \mathbf{Elim}(t'; \mathcal{D}.d_k; \vec{u'}; Q'_{d_1}, \ldots, Q'_{d_l}) \left\{f'_{c_1}, \ldots, f'_{c_{l'}}\right\} : Q_{d_k}\ \vec{u}\ \vec{m}\ t\end{array}}$$

Figure A.2: Inductive types and eliminators

- All inductive types in the block appear only strictly positively in constructors. In other words, for all $c \in \mathsf{dom}(\Delta_C)$, $\mathsf{Pos}_{\mathsf{dom}(\Delta_I)}(\Delta_C(c))$ where $\mathsf{Pos}_S(t)$ is determined by the following rules:

$$\frac{u \text{ does not appear in } \vec{A} \text{ or } \vec{t} \text{ for all } u \in S \qquad B \in S}{\mathsf{Pos}^{\dagger}_S(\mathbf{\Pi} \vec{x} : \vec{A}.\, B\ \vec{t})}$$

$$\frac{u \text{ does not appear in } \vec{t} \text{ for all } u \in S \qquad B \in S}{\mathsf{Pos}_S(B\ \vec{t})}$$

$$\frac{\mathsf{Pos}^{\dagger}_S(A) \qquad \mathsf{Pos}_S(B)}{\mathsf{Pos}_S(A \to B)}$$

$$\frac{u \text{ does not appear in } \vec{A} \text{ for all } u \in S \qquad \mathsf{Pos}_S(B)}{\mathsf{Pos}_S(\mathbf{\Pi} \vec{x} : \vec{A}.\, B)}$$

Inductive types and constructors Rules IND-TYPE and IND-CONSTR indicate, respectively, the type of inductive types and constructs in a block of mutually inductive types. The type of an inductive type of a block is exactly the same as declared in the block. The type of constructors of a block is determined by the type declared in the block except that inductive types in block are replaced by their proper (global) names.

Example A.3.1. *The following is the definition natural numbers in* PCIC.

$$\mathcal{N} \triangleq \mathbf{Ind}_0\{nat : \mathtt{Type}_0 := zero : nat, succ : nat \to nat\}$$

$$nat \triangleq \mathcal{N}.nat$$

$$zero \triangleq \mathcal{N}.zero$$

$$succ \triangleq \mathcal{N}.succ$$

Example A.3.2. *The following is the definition of universe polymorphic lists (for level i) in* PCIC.

$$\mathcal{L}_i \triangleq \mathbf{Ind}_1\{list : \mathbf{\Pi} A : \mathtt{Type}_i.\, \mathtt{Type}_i := nil : \mathbf{\Pi} A : \mathtt{Type}_i.\, list\ A,$$

$$cons : \mathbf{\Pi} A : \mathtt{Type}_i.\, A \to list\ A \to list\ A\}$$

$$list_i \triangleq \mathcal{L}_i.list$$

$$nil_i \triangleq \mathcal{L}_i.nil$$

$$cons_i \triangleq \mathcal{L}_i.cons$$

Example A.3.3. *Definition of a finitely branching tree as a mutual inductive block in* PCIC.

$$\begin{aligned}&\mathbf{Ind}_0\{FTree : \mathtt{Type}_0, Forest : \mathtt{Type}_0 :=\\ &\quad leaf : FTree, node : Forset \to FTree,\\ &\quad Fnil : Forest, Fcons : FTree \to Forest \to Forest\}\end{aligned}$$

Eliminators The term $\mathbf{Elim}(t; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}$ is the elimination of the term t (which is of type d_k (in some inductive block $\mathcal{D}$) applied to parameters $\vec{u}$) where the result of elimination of inductive types in the block, i.e., motives of eliminations, is given by $\vec{Q}$ and $\vec{f}$ are functions for elimination of terms constructed using particular constructors. The term, above, has type $Q_{d_k}\ \vec{u}\ \vec{m}\ t$ if t has type $d_k\ \vec{u}\ \vec{m}$.

Each case-eliminator f_{c_i} is the recipe for generating a term of the appropriate type (according to the corresponding motive) out of arguments of the constructor c_i under the assumption that all (mutually) recursive arguments are already appropriately eliminated. This is perhaps best seen in the rule IOTA below which describes the judgemental equality corresponding to the (intended) operational behavior of eliminators.

The function $\xi^{\vec{Q}}_{\mathcal{D}}(c_i, \Delta_C(c_i))$ ascribes a type to the case-eliminator f_{c_i} in the manner explained above. That is, $\xi^{\vec{Q}}_{\mathcal{D}}(c_i, \Delta_C(c_i))$ is a function type that given arguments of the constructor c_i (and their eliminated version if they are (mutually) recursive arguments) produces a term of the appropriate type according to the motives. It is formally defined as follows by recursion on derivation of $\mathsf{Pos}_{\mathsf{dom}(\Delta_I)}(\Delta_C(c_i))$:

If $P \equiv \mathbf{\Pi}\vec{x} : \vec{A}.\, d_i\ \vec{m}$ and we have $\mathsf{Pos}^{\dagger}_{\mathsf{dom}\Delta_I}(P)$ and $\mathsf{Pos}_{\mathsf{dom}\Delta_I}(B)$

$$\xi^{\vec{Q}}_{\mathcal{D}}(t, P \to B) \triangleq \mathbf{\Pi}p : P.\, (\mathbf{\Pi}\vec{x} : \vec{A}.\, Q_{d_i}\ m\ (p\ \vec{x})) \to \xi^{\vec{Q}}_{\mathcal{D}}(t\ p, B)$$

Otherwise,

$$\xi^{\vec{Q}}_{\mathcal{D}}(t, \mathbf{\Pi}x : A.\, B) \triangleq \mathbf{\Pi}x : A.\, \xi^{\vec{Q}}_{\mathcal{D}}(t\ x, B)$$

Otherwise, if $d \in \mathsf{dom}(\Delta_I)$

$$\xi^{\vec{Q}}_{\mathcal{D}}(t, d\ \vec{m}) \triangleq Q_d\ \vec{m}\ t$$

Example A.3.4. *The following is the definition of the recursive function corresponding to the principle of mathematical induction on natural numbers in* PCIC.

$$nat\_ind \triangleq \lambda P : nat \to \texttt{Prop}.\, \lambda o : P\ zero.\, \lambda stp : \mathbf{\Pi} x : nat.\ P\ x \to P\ (succ\ x).$$
$$\lambda n : nat.\, \mathbf{Elim}(n; nat; \mathsf{nil}; P)\ \{o, stp\}$$

The term nat_ind above has the type

$$\mathbf{\Pi} P : nat \to \texttt{Prop}.\, (P\ Z) \to (\mathbf{\Pi} x : nat.\, P\ x \to P\ (S\ x)) \to \mathbf{\Pi} n : nat.\, P\ n$$

Example A.3.5. *The following is the definition the recursive function to add two natural numbers in* PCIC.

$$add \triangleq \lambda n : nat.\, \lambda m : nat.$$
$$\mathbf{Elim}(n; nat; \mathsf{nil}; \lambda \_ : nat.\, nat)\ \{m, \lambda \_ : nat.\, \lambda y : nat.\, succ\ y\}$$

Example A.3.6. *The following is the definition the polymorphic and universe polymorphic recursive function to compute the length of a list in* PCIC.

$$length \triangleq \lambda A : \texttt{Type}_i.\, \lambda l : list_i\ A.\, \mathbf{Elim}(l; list; A; \lambda \_ : list_i.\, nat)$$
$$\{\lambda B : \texttt{Type}_i.\, zero, \lambda B : \texttt{Type}_i.\, \lambda x : B.\, \lambda y : list_i\ B.\, \lambda z : nat.\, succ\ z\}$$

A.4 Judgemental equality

The main typing rules for judgemental equality, except for those related to cumulativity, are presented in Figure A.3.

The rules Eq-ref, Eq-sym and Eq-trans make the judgemental equality an equivalence relation. The rule Beta corresponds to the operational rule for β-reduction in lambda calculus. The rules Delta and Zeta correspond to the operation of expansion of global definitions (defined in the context) and let-bindings, respectively. The rule Eta corresponds to η-expansion for (dependent) functions.

The rule Iota corresponds to the operation of reduction of recursive functions and case analysis (in systems featuring these, e.g., Coq itself) and that of eliminators as in our case. Notice that this rule only applies if the term on which elimination is being performed is a constructor applied to some terms[1].

[1] This restriction is indeed necessary in Coq so as to guarantee strong normalization.

$$\textsc{Eq-ref}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t \simeq t : A} \qquad \textsc{Eq-sym}\ \frac{\Gamma \vdash t \simeq t' : A}{\Gamma \vdash t' \simeq t : A} \qquad \textsc{Eq-trans}\ \frac{\Gamma \vdash t \simeq t' : A \qquad \Gamma \vdash t \simeq t'' : A}{\Gamma \vdash t \simeq t'' : A}$$

$$\textsc{Beta}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma, x : A \vdash B : s \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.\, M)\ N \simeq M[N/x] : B[N/x]}$$

$$\textsc{Delta}\ \frac{\mathcal{WF}(\Gamma) \qquad x := t : A \in \Gamma}{\Gamma \vdash x \simeq t : A} \qquad \textsc{Eta}\ \frac{\Gamma \vdash t : \mathbf{\Pi} x : A.\, B}{\Gamma \vdash t \simeq \lambda x : A.\, t\ x : \mathbf{\Pi} x : A.\, B}$$

$$\textsc{Zeta}\ \frac{\Gamma \vdash t : A \qquad \Gamma, x := t : A \vdash u : B}{\Gamma \vdash (\mathbf{let}\, x := t : A\, \mathbf{in}\, u) \simeq u[t/x] : B[t/x]}$$

$$\textsc{Iota}\ \frac{\begin{array}{c}\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \qquad \Gamma \vdash \mathcal{D}' \preceq^{\dagger} \mathcal{D} \\ \mathrm{dom}(\Delta_I) = \{d_1, \ldots, d_l\} \qquad \mathrm{dom}(\Delta_C) = \{c_1, \ldots, c_{l'}\} \\ (\Gamma \vdash Q_{d_i} : \mathbf{\Pi} \vec{x} : \vec{A}.\, (d_i\ \vec{x}) \to s' \text{ where} \\ \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{x} : \vec{A}.\, s \text{ for all } 1 \leq i \leq l) \\ \Gamma \vdash \mathcal{D}'.c_j\ \vec{a} : \mathcal{D}.d_k\ \vec{u}\ \vec{m} \qquad \mathsf{len}(\vec{u}) = n \qquad 1 \leq j \leq l' \\ \Gamma \vdash f_{c_i} : \xi_{\mathcal{D}}^{\vec{Q}}(\mathcal{D}.c_i, \Delta_C(c_i)) \text{ for all } 1 \leq i \leq l'\end{array}}{\begin{array}{c}\Gamma \vdash \mathbf{Elim}(\mathcal{D}'.c_j\ \vec{a}; \mathcal{D}.d_k; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\} \simeq \\ \mu_{\mathcal{D}}^{\vec{Q};\vec{f}}(f_{c_j}; \vec{a}; \Delta_C(c_j)) : Q_{d_k}\ \vec{u}\ \vec{m}\ (\mathcal{D}'.c_j\ \vec{a})\end{array}}$$

Figure A.3: The main judgemental equality rules

This rule specifies that, as expected, the eliminator, applied to a term that is constructed out of a constructor (of the corresponding type or any of its sub-types, see Remark A.4.1 below) applied to some terms, is equivalent to the corresponding case-eliminator applied arguments of the constructor, after (mutually) recursive arguments are appropriately eliminated. This is exactly what the recursor $\mu_{\mathcal{D}}^{\vec{Q};\vec{f}}$ does, applying arguments of the constructor to the corresponding case-eliminator after eliminating (mutually) recursive arguments as necessary. Recursors are defined as follows by recursion on the derivation of $\mathsf{Pos}_{\mathrm{dom}(\Delta_I)}(\Delta_C(c_i))$:

If $P \equiv \mathbf{\Pi} \vec{x} : \vec{A}.\, d_i\ \vec{m}$ and we have $\mathsf{Pos}^{\dagger}_{\mathsf{dom}\Delta_I}(P)$ and $\mathsf{Pos}_{\mathsf{dom}\Delta_I}(B)$

$$\mu_{\mathcal{D}}^{\vec{Q};\vec{f}}(t; b, \vec{a}; P \to B) \triangleq \mu_{\mathcal{D}}^{\vec{Q};\vec{f}}$$

$$\left(t\ b\ \left(\lambda \vec{x} : \vec{A}.\, \mathbf{Elim}(b\ \vec{x}; \mathcal{D}.d_i; \vec{Q})\left\{\vec{f}\right\}\right); \vec{a}; B\right)$$

Otherwise,

$$\mu_{\mathcal{D}}^{\vec{Q};\vec{f}}(t; b, \vec{a}; \mathbf{\Pi} x : A.\, B) \triangleq \mu_{\mathcal{D}}^{\vec{Q};\vec{f}}(t\ b; \vec{a}; B)$$

Otherwise, if $d \in \mathsf{dom}(\Delta_I)$

$$\mu_{\mathcal{D}}^{\vec{Q};\vec{f}}(t; \mathsf{nil}; d\ \vec{m}) \triangleq t$$

Remark A.4.1 (IOTA and subtyping of cumulativity for inductive tpyes)**.** *We, in addition, stipulate here that the eliminators can eliminate terms constructed using the corresponding constructor of any inductive type that is a sub-type of the inductive type for which the elimination is specified. Note that in the Coq implementation of cumulativity for inductive types, cumulativity is only considered for different instances of the same inductive type at different universe levels. That is, only for two instances of the same universe polymorphic inductive type. In those settings, the inductive types being sup-types of one another are instance of the same inductive types and the eliminators, and in general the operational semantics, simply ignore the universes in terms. Here, we have to consider this side condition to achieve a similar result.*

A.5 Conversion/Cumulativity

Figures A.4 shows the conversion/cumulativity rules of PCIC.

The core of these rules is the rule CUM. It states that whenever a term t has type A and the conversion/cumulativity relation $A \preceq B$ holds, then t also has type B. The rule EQ-CUM says that two judgementally equal (convertible) types M and M' are in conversion/cumulativity relation $M \preceq M'$. The rules PROP-IN-TYPE and CUM-TYPE specify the order on the hierarchy of sorts. The rule CUM-PROD states the conditions for conversion/cumulativity between two (dependent) function types. Note in this rule that functions are *not* contravariant in their domain type. This is also the case in Coq. This condition is crucial for the construction of our set-theoretic model as set-theoretic functions (*i.e.*, functional relations) are not contravariant.

$$\textsc{Prop-in-Type}\quad \Gamma \vdash \texttt{Prop} \preceq \texttt{Type}_i$$

$$\textsc{Cum-Type}\quad \frac{i \leq j}{\Gamma \vdash \texttt{Type}_i \preceq \texttt{Type}_j}$$

$$\textsc{Cum-trans}\quad \frac{\Gamma \vdash M \preceq N \qquad \Gamma \vdash N \preceq O}{\Gamma \vdash M \preceq O}$$

$$\textsc{Cum-weaken}\quad \frac{\Gamma \vdash M \preceq N \qquad \Gamma \vdash O : s \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x : O \vdash M \preceq N}$$

$$\textsc{Cum-Prod}\quad \frac{\Gamma \vdash A_1 \simeq B_1 : s \qquad \Gamma, x : A_1 \vdash A_2 \preceq B_2}{\Gamma \vdash \mathbf{\Pi} x : A_1 . A_2 \preceq \mathbf{\Pi} x : B_1 . B_2}$$

$$\textsc{Cum-eq-L}\quad \frac{\Gamma \vdash M \simeq N : s \qquad \Gamma \vdash N \preceq O}{\Gamma \vdash M \preceq O}$$

$$\textsc{Cum-eq-R}\quad \frac{\Gamma \vdash M \preceq N \qquad \Gamma \vdash N \simeq O : s}{\Gamma \vdash M \preceq O}$$

$$\textsc{Cum}\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash t : B}$$

$$\textsc{Cum-eq}\quad \frac{\Gamma \vdash t \simeq t' : A \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash t \simeq t' : B}$$

$$\textsc{Eq-Cum}\quad \frac{\Gamma \vdash M \simeq M' : s}{\Gamma \vdash M \preceq M'}$$

Figure A.4: Cumulativity

Appendix B

The set theoretic model of pCuIC

B.1 Set-theoretic background

In this section we shortly explain the set-theoretic axioms and constructions that form the basis of our model. We assume that the reader is familiar with the ZFC set theory. This is very similar to the theory that Lee and Werner (2011) use as the basis for their model. In particular, we use Zermelo-Fraenkel set theory with the axiom of choice (ZFC) together with an axiom that there is a countably infinite strictly increasing hierarchy of uncountable strongly inaccessible cardinals. In particular, we assume that we have a hierarchy of *strongly inaccessible* cardinals $\kappa_0, \kappa_1, \kappa_2, \ldots$ where $\kappa_0 > \omega$.

B.1.1 Von Neumann cumulative hierarchy and models of ZFC

The von Neumann cumulative hierarchy is a sequence of sets (indexed by ordinal numbers) that is cumulative. That is, each set in the hierarchy is a subset of the all sets after it. These sets are also referred to as von Neumann universes. This hierarchy is defined as follows for ordinal number α:

$$\mathcal{V}_\alpha \triangleq \bigcup_{\beta \in \alpha} \mathcal{P}\left(\mathcal{V}_\beta\right)$$

It is well-known (Drake, 1974) that whenever α is a strongly inaccessible cardinal number of a cardinality strictly greater than ω, as is the case for $\kappa_0, \kappa_2, \ldots, \mathcal{V}_\alpha$

is a model for ZFC. The von Neumann universe $\mathcal{V}_\omega$ satisfies all axioms of ZFC except for the axiom of infinity. This is due to the fact that all sets in $\mathcal{V}_\omega$ are *finite*. This is why we have assumed that our hierarchy of strngly inaccessible cardinals starts at one strictly greater than ω. In particular, if A and B are two sets in $\mathcal{V}_\kappa$ then $B^A \in \mathcal{V}_\kappa$ where B^A is the set of all functions from B to A. This allows us to use von Neumann universes to model the predicative PCUIC universes. For more details about strongly inaccessible cardinals and von Neumann universes refer to Drake (1974).

B.2 Rule sets and fixpoints: inductive constructions in set theory

Following Lee and Werner (2011), who follow Dybjer (1991) and Aczel (1999), we use inductive definitions (in set theory) constructed through rule sets to model inductive types. Here, we give a very short account of rule sets for inductive definitions. For further details refer to Aczel (1977).

A pair (A, a) is a *rule* based on a set U where $A \subseteq U$ is the set of premises and $a \in U$ is the conclusion. We usually write $\frac{A}{a}$ for a rule (A, a). A *rule set* based on U is a set Φ of rules based on U. We say a set $X \subseteq U$ is Φ-*closed*, $closed_\Phi(X)$ for a U based rule set Φ if we have:

$$closed_\Phi(X) \triangleq \forall \frac{A}{a} \in \Phi.\ A \subseteq X \Rightarrow a \in X$$

The operator $\mathcal{O}_\Phi$ corresponding to a rule set Φ is the operation of collecting all conclusions for a set whose premises are available in that set. That is,

$$\mathcal{O}_\Phi(X) \triangleq \left\{ a \middle| \frac{A}{a} \in \Phi \wedge A \subseteq X \right\}$$

Hence, a set X is Φ-closed if $\mathcal{O}_\Phi(X) \subseteq X$. Notice that $\mathcal{O}_\Phi$ is a monotone function on $\mathcal{P}(U)$ which is a complete lattice. Therefore, for any U based rule set Φ, the operator $\mathcal{O}_\Phi$ has a least fixpoint, $\mathcal{I}(\Phi) \subseteq U$:

$$\mathcal{I}(\Phi) \triangleq \bigcap \{X \subseteq U | closed_\Phi(X)\}$$

We define by transfinite recursion a sequence, indexed by ordinal numbers $\mathcal{O}_\Phi^\alpha$ for an ordinal number α:

$$\mathcal{O}_\Phi^\alpha \triangleq \bigcup_{\beta \in \alpha} \left(\mathcal{O}_\Phi^\beta \cup \mathcal{O}_\Phi(\mathcal{O}_\Phi^\beta) \right)$$

Obviously, for $\beta \leq \alpha$ we have $\mathcal{O}_\Phi^\beta \subseteq \mathcal{O}_\Phi^\alpha$.

Theorem B.2.1 (Aczel (1977))**.** *For any rule set* Φ *there exists an ordinal number* $ClOrd(\Phi)$ *called the* closing ordinal *of* Φ *such that it is the smallest ordinal number for which we have* $\mathcal{I}(\Phi) = \mathcal{O}_\Phi^{ClOrd(\Phi)}$ *is the least fixpoint of* $\mathcal{O}_\Phi$*. In other words,* $\mathcal{O}_\Phi^{ClOrd(\Phi)+1} = \mathcal{O}_\Phi(\mathcal{O}_\Phi^{ClOrd(\Phi)}) = \mathcal{O}_\Phi^{ClOrd(\Phi)}$*.*

Lemma B.2.2 (Aczel (1977))**.** *Let* Φ *be a rule set based on some set* U *and* β *a* regular *cardinal such that for every rule* $\frac{A}{a} \in \Phi$ *we have that* $|A| < \beta$ *then*

$$ClOrd(\Phi) \leq \beta$$

In other words, $\mathcal{O}_\Phi^\beta$ *is the least fixpoint of* $\mathcal{O}_\Phi$*.*

B.2.1 Fixpoints of large functions

A set theoretic constructions is called *large* with respect to a set theoretic (von Neumann) universe if it does not belong to that universe. As we shall see, the functions that we will consider for interpreting of inductive types (operators of certain rule sets) are indeed large. That is, they map subsets of the universe to subsets of the universe. As a result, the fixpoint of these functions might not have a fixpoint within the universe in question as the universe with the subset relation on it is not a complete lattice. The following lemmas show that under certain conditions, the fixpoint of the function induced by rule sets does exist in the desired universe. We will use this lemma to show that the interpretations of inductive types do indeed fall in the universe that they are supposed to.

Lemma B.2.3. *Let* Φ *be a rule set based on the set-theoretic universe* $\mathcal{V}$ *and* $\alpha \in \mathcal{V}$ *be a cardinal number such that for all* $\frac{A}{a} \in \Phi$ *we have* $|A| \leq \alpha$*. Then,*

$$ClOrd(\Phi) \in \mathcal{V}$$

Proof. By Lemma B.2.2, it suffices to show that there is a regular cardinal $\beta \in \mathcal{V}$ such that $|A| < \beta$ for any $\frac{A}{a} \in \Phi$. Take β to be $\aleph_{\alpha+1}$. By the fact that $\mathcal{V}$ is a model of ZFC we know that $\aleph_{\alpha+1} \in \mathcal{V}$. By the fact that $\alpha < \aleph_{\alpha+1}$ we know that $|A| < \aleph_{\alpha+1}$ for any $\frac{A}{a} \in \Phi$. It is well known that under axioms of ZFC (this crucially uses axiom of choice) $\aleph_{\alpha+1}$ is a regular cardinal for any ordinal number α – see Drake (1974) for a proof. This concludes our proof. □

Lemma B.2.4. *Let* Φ *be a rule set based on the set-theoretic universe* $\mathcal{V}$ *and* $\alpha \in \mathcal{V}$ *be a cardinal number such that for all* $\frac{A}{a} \in \Phi$ *we have* $|A| \leq \alpha$*. Then,*

$$\mathcal{I}(\Phi) \in \mathcal{V}$$

Proof. By Lemma B.2.2 we know that $\mathcal{I}(\Phi) = \mathcal{O}_{\Phi}^{ClOrd(\Phi)}$. We know that $\mathcal{O}_{\Phi}^{ClOrd(\Phi)} \in \mathcal{V}$ as it is constructed by transfinite recursion up to $ClOrd(\Phi)$ and that we crucially know that $ClOrd(\Phi) \in \mathcal{V}$ by Lemma B.2.3. More precisely, this can be shown using transfinite induction up to $ClOrd(\Phi)$ showing that every stage belongs to $\mathcal{V}$. Notice that it is crucial for an ordinal to belong to the universe in order for transfinite induction to be valid. □

B.2.2 The use of the axiom of choice

The only place in this work that we make use of axiom of choice is in the proof of Lemma B.2.3. We use this axiom to show the following statement which we could have alternatively taken as a (possibly) weaker axiom.

> In any von Neumann universe $\mathcal{V}$ for any cardinal number α there is a *regular* cardinal β such that $\alpha < \beta$.

Note that his statement is independent of ZF and certain axiom, e.g., the axiom of choice as we have taken here, needs to be postulated. This is due to the well-known fact proven by Gitik (1980) that under the assumption of existence of strongly compact cardinals, any uncountable cardinal is singular!

B.2.3 Modeling the impredicative sort Prop: trace encoding

One of the challenges in constructing a model for a system like PCUIC is treatment of an impredicative proof-irrelevant sort Prop. This can be done by simply modeling Prop as the set $\{\emptyset, \{\emptyset\}\}$ where provable propositions are modeled as $\{\emptyset\}$ and non-provable propositions as $\emptyset$. This however, will only work where we don't have the cumulativity relation between Prop and $\mathtt{Type}_i$. In presence of such cumulativity relations, such a naïve treatment of Prop breaks interpretation of the (dependent) function spaces as sets of functions. The following example should make the issue plain.

Example B.2.5 (Werner (1997))**.** *Let's consider the interpretation of the term* $I \equiv \lambda(P : \mathtt{Type}_0).\, P \to P$. *In this case, the semantics of* $[\![I\ True]\!]$ *will be* $\{\emptyset\}$ *or* $\{\emptyset\}^{\{\emptyset\}}$ *depending on whether* $True : \mathtt{Prop}$ *or* $True : \mathtt{Type}_0$ *is considered, respectively. And hence we should have* $\{\emptyset\} = \{\emptyset\}^{\{\emptyset\}}$ *which is not the case, even though the two sets are isomorphic (bijective).*

In order to circumvent this issue, we follow Lee and Werner (2011), who in turn follow Aczel (1999), in using the method known as *trace encoding* for representation of functions.

Definition B.2.6 (Trace encoding)**.** *The following two functions,* Lam *and* App*, are used for trace encoding and application of trace encoded functions respectively.*

$$\mathsf{Lam}(f) \triangleq \bigcup_{(x,y)\in f} (\{x\} \times y)$$

$$\mathsf{App}(f, x) \triangleq \{y | (x, y) \in f\}$$

Lemma B.2.7. *Let $f : X \to Y$ be a set theoretic function then for any $x \in X$ we have*

$$\mathsf{App}(\mathsf{Lam}(f), x) = f(x)$$

Note that using the trace encoding technique the problem mentioned in Example B.2.5 is not present anymore. That is, we have:

$$\left\{\mathsf{Lam}(f) \middle| f \in \{\emptyset\}^{\{\emptyset\}}\right\} = \{\emptyset\}$$

Lemma B.2.8 (Aczel (1999))**.** *Let A be a set and assume the set $B(x) \subseteq 1$ for $x \in A$.*

1. $\{\mathsf{Lam}(f) | f \in \Pi x \in A.\, B(x)\} \subseteq 1$
2. $\{\mathsf{Lam}(f) | f \in \Pi x \in A.\, B(x)\} = 1$ *iff* $\forall x \in A.\, B(x) = 1$

B.3 The set-theoretic model of pCuIC and its soundness

We construct a model for PCUIC by interpreting predicative universes using von Neumann universes and `Prop` as $\{0, 1\} = \{\emptyset, \{\emptyset\}\}$. We use the trace encoding technique presented earlier for (dependent) function types. We will construct the interpretation of inductive types and their eliminators using rule sets for inductive definitions in set theory. We shall first define a *size* function on terms, typing contexts, and pairs of a context and a term which we write as $\mathit{size}(\Gamma \vdash t)$. We will then define the interpretation of typing contexts and terms (in appropriate context) by well-founded recursion on their size.

Definition B.3.1. *We define a function called size on terms, contexts, declarations and pairs consisting of a context and a term (which we write as $\mathit{size}(\Gamma \vdash t)$) mutually recursively as follows:*

Size for typing contexts and declarations

$$size(\cdot) \triangleq 1/2$$

$$size(\Gamma, x : A) \triangleq size(\Gamma) + size(A)$$

$$size(\Gamma, x := t : A) \triangleq size(\Gamma) + size(t) + size(A)$$

$$size(\Gamma, \mathbf{Ind}_n \{\Delta_I := \Delta_C\}) \triangleq size(\Gamma) + size(\mathbf{Ind}_n \{\Delta_I := \Delta_C\})$$

Size for term

$$size(\texttt{Prop}) \triangleq 1$$

$$size(\texttt{Type}_i) \triangleq 1$$

$$size(x) \triangleq 1$$

$$size(\mathbf{\Pi} x : A.\, B) \triangleq size(A) + size(B) + 1$$

$$size(\lambda x : A.\, t) \triangleq size(A) + size(t) + 1$$

$$size(t\ u) \triangleq size(t) + size(u) + 1$$

$$size(\mathbf{let}\, x := t : A\, \mathbf{in}\, u) \triangleq size(t) + size(u) + size(A) + 1$$

$$size(\mathcal{D}.z) \triangleq size(\mathcal{D})$$

$$size(\mathbf{Elim}(t; \mathcal{D}.d_i; \vec{u}; \vec{Q}) \left\{ \vec{f} \right\}) \triangleq size(t) + size(\mathcal{D}) + \sum_i size(u_i)$$

$$\sum_i size(Q_i) + \sum_i size(f_i) + 1$$

Size for pairs consisting of a context and a term

$$size(\mathbf{Ind}_n \{\Delta_I := \Delta_C\}) \triangleq \sum_{d \in \mathsf{dom}(\Delta_I)} size(\Delta_I(d)) +$$

$$\sum_{c \in \mathsf{dom}(\Delta_C)} size(\Delta_C(c)) + 1$$

Size for judgements

$$size(\Gamma \vdash t) \triangleq size(\Gamma) + size(t) - 1/2$$

$$size(\Gamma \vdash \mathbf{Ind}_n \{\Delta_I := \Delta_C\}) \triangleq size(\Gamma) + size(\mathbf{Ind}_n \{\Delta_I := \Delta_C\}) - 1/2$$

In the definition above, which is similar to that by Lee and Werner (2011) and Miquel and Werner (2003), the $\pm 1/2$ is add to make sure $size(\Gamma \vdash t) < size(\Gamma, x : t)$ and that $size(\Gamma) < size(\Gamma \vdash t)$.

Definition B.3.2 (The model)**.** *We define the interpretations for contexts and terms by well-founded recursion on their sizes.*

Interpretation of contexts

$$[\![\cdot]\!] \triangleq \{\mathsf{nil}\}$$

$$[\![\Gamma, x : A]\!] \triangleq \left\{ \gamma, a \middle| \gamma \in [\![\Gamma]\!] \wedge [\![\Gamma \vdash A]\!]_\gamma \downarrow \wedge a \in [\![\Gamma \vdash A]\!]_\gamma \right\}$$

$$[\![\Gamma, x := t : A]\!] \triangleq \left\{ \gamma, a \middle| \begin{array}{l} \gamma \in [\![\Gamma]\!] \wedge [\![\Gamma \vdash A]\!]_\gamma \downarrow \wedge [\![\Gamma \vdash t]\!]_\gamma \downarrow \wedge \\ a = [\![\Gamma \vdash t]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma \end{array} \right\}$$

$$[\![\Gamma, \mathbf{Ind}_n \{\Delta_I := \Delta_C\}]\!] \triangleq [\![\Gamma]\!] \text{ if } [\![\Gamma \vdash \mathbf{Ind}_n \{\Delta_I := \Delta_C\}]\!]_\gamma \downarrow \text{ for all } \gamma \in [\![\Gamma]\!]$$

Above, we assume that $x \notin \mathsf{dom}(\Gamma)$, otherwise, both $[\![\Gamma, x : A]\!]$ and $[\![\Gamma, x := t : A]\!]$ are undefined.

Interpretation of terms

$$[\![\Gamma \vdash \texttt{Prop}]\!]_\gamma \triangleq \{\emptyset, \{\emptyset\}\}$$

$$[\![\Gamma \vdash \texttt{Type}_i]\!]_\gamma \triangleq \mathcal{V}_{\kappa_i}$$

$$[\![\Gamma \vdash x]\!]_{\vec{a}} \triangleq a_{\mathsf{len}(\Gamma_1)-l} \qquad \text{if } \Gamma = \Gamma_1, x : A, \Gamma_2 \text{ and } x \notin \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \text{ and } l = \mathsf{len}(Inds(\Gamma_1))$$

$$[\![\Gamma \vdash \mathbf{\Pi} x : A.\, B]\!]_\gamma \triangleq \left\{ \mathsf{Lam}(f) \middle| f : \Pi a \in [\![\Gamma \vdash A]\!]_\gamma .\ [\![\Gamma, x : A \vdash B]\!]_{\gamma,a} \right\}$$

$$[\![\Gamma \vdash \lambda x : A.\, t]\!]_\gamma \triangleq \mathsf{Lam}\left(\left\{ (a, [\![\Gamma, x : A \vdash t]\!]_{\gamma,a}) \middle| a \in [\![\Gamma \vdash A]\!]_\gamma \right\} \right)$$

$$[\![\Gamma \vdash t\ u]\!]_\gamma \triangleq \mathsf{App}([\![\Gamma \vdash t]\!]_\gamma, [\![\Gamma \vdash u]\!]_\gamma)$$

$$[\![\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u]\!]_\gamma \triangleq [\![\Gamma, x := t : A \vdash u]\!]_{\gamma, [\![\Gamma \vdash u]\!]_\gamma}$$

Interpretation of inductive types, constructors and eliminators is defined below

B.3.1 Modeling inductive types

We define the interpretation for inductive blocks by constructing a rule set which will interpret the whole inductive block. We will define interpretation of individual inductive types and constructors of the block based on the fixpoint of this rule set.

Remark B.3.3. *In the reset of this chapter, we assume for the construction of the model for inductive types, constructors and eliminators, that the parameters can be heterogeneous. The only lemma that we cannot prove for heterogeneous parameters is Lemma B.3.8 below. This is the reason why in our system we have assumed that parameters are uniform, i.e, they are fixed for the whole block. Proving an analogue of Lemma B.3.8 for inductive types with heterogeneous parameters is one of the future works of our work.*

Definition B.3.4 (Interpretation of inductive types)**.** *Let us assume that* $\mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\}$ *is an arbitrary but fixed inductive block such that* $\Delta_I = d_0 : A_0, \ldots d_l : A_l$ *and* $\Delta_C = c_0 : T_0, \ldots c_{l'} : T_{l'}$*. Here, we assume* $A_i \equiv \mathbf{\Pi} \vec{p} : \vec{P}. \mathbf{\Pi} \vec{x} : \vec{B_i}. s$ *for some sequence of types* $\vec{B}$ *and some sort* s*. Furthermore,* $\vec{P}$ *are parameters of the inductive block. The type of constructors are of the following form:* $T_k \equiv \mathbf{\Pi} \vec{p} : \vec{P}. \mathbf{\Pi} \vec{x} : \vec{V_k}. d_{i_k}\ \vec{p}\ \vec{t_k}$ *for some* $\vec{t_k}$*. Notice that* $\vec{V_k}$ *is a sequence itself. That is, it is of the form* $\vec{V_k} \equiv V_{k,1}, \ldots, V_{k,\mathsf{len}(\vec{V_K})}$*. That is, each constructor* c_k*, takes a number of arguments, first parameters* $\vec{P}$ *and then some more* $\vec{V_k}$*. The strict positivity condition implies that for any non-parameter argument* $V_{k,i}$ *of a constructor* c_k*, either* $d \in \mathsf{dom}(\Delta_I)$ *does not appear in* $V_{k,i}$ *or we have* $V_{k,i} \equiv \mathbf{\Pi} \vec{x} : \overrightarrow{W_{k,i}}. d_{I_{k,i}}\ \vec{q}\ \vec{t}$ *where* $\mathsf{len}(\vec{q}) = \mathsf{len}(\vec{p})$*. That is, each argument of a constructor where an inductive type (of the same block) appears, is a (dependent) function with codomain being that inductive type. In this case, no inductive type of the block appears the domain(s) of the function,* $\overrightarrow{W_{k,i}}$*. Notice that the codomain of the function is an inductive type in the block but not necessarily of the same family as the one being defined – the parameters applied are* $\vec{q}$ *instead of* $\vec{p}$*! We write* $rec(V_{k,i})$ *if some inductive of the block appears in* $V_{k,i}$ *in which case it will be of the form just described.*

$$[\![\Gamma \vdash \mathcal{D}]\!]_\gamma \triangleq \mathcal{I}(\Phi^\gamma_{\Gamma,\mathcal{D}})$$

$$\Phi^\gamma_{\Gamma,\mathcal{D}} \triangleq \bigcup_{d_i \in \mathsf{dom}(\Delta_I)} \bigcup_{c_k \in \mathsf{Constrs}(d_i)} \left\{ \frac{\Psi_{d_i,c_k}}{\psi_{d_i,c_k}} \;\middle|\; \begin{array}{l} \vec{a} \in [\![\Gamma \vdash \vec{P}]\!]_\gamma, \\ \vec{m} \in [\![\Gamma, \vec{p} : \vec{P}, \vec{d'} : \vec{s} \vdash \vec{V'_k}]\!]_{\gamma, \vec{a}, \vec{b}} \end{array} \right\}$$

Where, $\overrightarrow{d'}$ *is the sequence of* $d'_{I_{k,j}}$*'s occurring within* $\overrightarrow{V'_k}$ *below and* $s_i \equiv A_{d_i}$ *is the arity of the inductive type* d_i*,* $\overrightarrow{b} \in \left[\!\left[\Gamma, \overrightarrow{p} : \overrightarrow{P} \vdash \overrightarrow{s}\right]\!\right]_{\gamma, \overrightarrow{a}}$ *and,*

$$\psi_{d_i,c_k} \triangleq \left\langle i; \overrightarrow{a}; \left[\!\left[\Gamma, \overrightarrow{p} : \overrightarrow{P}, \overrightarrow{d'} : \overrightarrow{s}, \overrightarrow{x} : \overrightarrow{V'_k} \vdash \overrightarrow{t_k}\right]\!\right]_{\gamma, \overrightarrow{a}, \overrightarrow{b}, \overrightarrow{m}} ; \langle k; \overrightarrow{m}\rangle \right\rangle$$

$$\Psi_{d_i,c_k} \triangleq \bigcup_{rec(V_{k,j})} \left\{ \left\langle I_{k,j}; \left[\!\left[\begin{array}{l} \Gamma_{k,j}, \\ \overrightarrow{y} : \overrightarrow{W_{k,j}} \end{array} \vdash \overrightarrow{q} \right]\!\right]_{\substack{\gamma, \overrightarrow{a}. \overrightarrow{b}, \\ \overrightarrow{e}, \overrightarrow{u}}} ; \left[\!\left[\begin{array}{l} \Gamma_{k,j}, \\ \overrightarrow{y} : \overrightarrow{W_{k,j}} \end{array} \vdash \overrightarrow{t} \right]\!\right]_{\substack{\gamma, \overrightarrow{a}, \overrightarrow{b}, \\ \overrightarrow{e}, \overrightarrow{u}}} ; \overrightarrow{\mathsf{App}}(m_{I_{k,j}}, \overrightarrow{u}) \right\rangle \;\middle|\; \overrightarrow{u} \in \left[\!\left[\Gamma_{k,j} \vdash \overrightarrow{W_{k,j}}\right]\!\right]_{\gamma, \overrightarrow{a}, \overrightarrow{b}, \overrightarrow{e}} \right\}$$

$$\Gamma_{k,j} \triangleq \Gamma, \overrightarrow{p} : \overrightarrow{P}, \overrightarrow{d'} : \overrightarrow{s}, x_1 : V'_{k,1}, \ldots, x_{j-1} : V'_{k,j-1}$$

$$\overrightarrow{e} \triangleq m_1, \ldots m_{j-1}$$

For Ψ_k *we assume that* $V_{k,j} \equiv \mathbf{\Pi} \overrightarrow{y} : \overrightarrow{W_{k,j}}. d_{I_{k,j}}\ \overrightarrow{q}\ \overrightarrow{t}$*.*
The types $V'_{k,j}$ *are defined based on* $V_{k,j}$ *as follows:*

$$V'_{k,j} \triangleq \begin{cases} \mathbf{\Pi} \overrightarrow{x} : \overrightarrow{W_{k,i}}. d'_{I_{k,j}} & \text{if } rec(V_{k,j}) \text{ and} \\ & V_{k,i} \equiv \mathbf{\Pi} \overrightarrow{x} : \overrightarrow{W_{k,i}}. d_{I_{k,j}}\ \overrightarrow{q}\ \overrightarrow{t} \\ V_{k,j} & \text{otherwise} \end{cases}$$

We define the interpretation of the inductive types in the block as follows:

$$\left[\!\left[\Gamma \vdash \mathcal{D}.d_i\right]\!\right]_\gamma \triangleq \overrightarrow{\mathsf{Lam}}(f_{d_i})$$

$$f_{d_i}(\overrightarrow{a}, \overrightarrow{t}) \triangleq \left\{ \langle k; \overrightarrow{m}\rangle \middle| \langle i; \overrightarrow{a}; \overrightarrow{t}; \langle k; \overrightarrow{m}\rangle\rangle \in \left[\!\left[\Gamma \vdash \mathcal{D}\right]\!\right]_\gamma \right\}$$

$$\text{for } \overrightarrow{a}, \overrightarrow{t} \in \left[\!\left[\Gamma \vdash \overrightarrow{P}, \overrightarrow{B_i}\right]\!\right]_\gamma$$

We define the interpretation of the constructors in the block as follows:

$$\left[\!\left[\Gamma \vdash \mathcal{D}.c_k\right]\!\right]_\gamma \triangleq \overrightarrow{\mathsf{Lam}}(g_{c_k})$$

$$g_{c_k}(\overrightarrow{a}, \overrightarrow{m}) \triangleq \langle k; \overrightarrow{m}\rangle \text{ for } \overrightarrow{a}, \overrightarrow{m} \in \left[\!\left[\Gamma \vdash \overrightarrow{P}, \overrightarrow{V_k}\right]\!\right]_\gamma$$

Let us first discuss the intuitive construction of Definition B.3.4 above. In this definition the most important part is the interpretation of the inductive block. We have defined this as the fixpoint of the rule sets corresponding to constructors of the inductive type. This rule set basically spells out the following. Take, some set $d'_{I_{k,j}}$ in the universe as a candidate for the interpretation of j^{th} occurrence of inductive type $d_{I_{k,j}}$ in the k^{th} constructor. This candidate, $d'_{I_{k,j}}$, is taken to be a candidate element of the inductive type $d_{I_{k,j}}$ in case $\overrightarrow{W_{k,i}} = \mathsf{nil}$, i.e., intuitively, if the recursive occurrence *embeds* an element of $d_{I_{k,j}}$ in the type being constructed. On the other hand, if the recursive occurrence of $d_{I_{k,j}}$ is, intuitively, *embedding* a function with codomain $d_{I_{k,j}}$ into the type being constructed then $d'_{I_{k,j}}$ is to be understood as the codomain of the function being embedded by the constructor. In each of these two cases, we need to make sure the candidate element and or function is *correctly chosen*, i.e., we need to make sure that the element or the *range* of the function chosen is indeed in the interpretation of the inductive type. This is where the rule sets come to play, so to speak. The premise set Ψ_k makes sure that all candidate recursive occurrences are indeed correctly chosen. This is done by basically making sure that for any of the arguments of the function being embedded (here an element is treated as a function with no arguments!) the result of applying the candidate function to the arguments is indeed in the interpretation of the corresponding inductive type. Do notice that $\overrightarrow{\mathsf{App}}(a, \mathsf{nil}) = a$.

Example B.3.5 (Rule set for construction of natural numbers)**.**

$$\Phi^{\gamma}_{\Gamma,\mathcal{N}} = \left\{ \frac{\emptyset}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 0; \mathsf{nil}\rangle\rangle} \right\} \cup \left\{ \frac{\{\langle 0; \mathsf{nil}; \mathsf{nil}; a\rangle\}}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 1; a\rangle\rangle} \middle| a \in \mathcal{V}\kappa_0 \right\}$$

Example B.3.6 (Rule set for construction of lists)**.**

$$\Phi^{\gamma}_{\Gamma,\mathcal{L}_i} = \left\{ \frac{\emptyset}{\langle 0; A; \mathsf{nil}; \langle 0; \mathsf{nil}\rangle\rangle} \right\} \cup \left\{ \frac{\{\langle 0; A; \mathsf{nil}, b\rangle\}}{\langle 0; A; \mathsf{nil}; \langle 1; a, b\rangle\rangle} \middle| b \in \mathcal{V}\kappa_0 \wedge a \in [\![\Gamma \vdash A]\!]_{\gamma} \right\}$$

Lemma B.3.7. *Values for arguments of arities are uniquely determined by the values for arguments each constructor (including values for parameters) in* $\mathcal{O}^{\alpha}_{\Phi^{\gamma}_{\Gamma,\mathcal{D}}}$ *and in particular in* $[\![\Gamma \vdash \mathcal{D}]\!]_{\gamma}$*. That is, if*

$$\left\langle i; \vec{a}; \vec{t}; \langle k; \vec{m}\rangle \right\rangle, \left\langle i; \vec{a}; \vec{t'}; \langle k; \vec{m}\rangle \right\rangle \in \mathcal{O}^{\alpha}_{\Phi^{\gamma}_{\Gamma,\mathcal{D}}}$$

then, $\vec{t} = \vec{t'}$*. Analogously for* $[\![\Gamma \vdash \mathcal{D}]\!]_{\gamma}$ *we have that if*

$$\left\langle i; \vec{a}; \vec{t}; \langle k; \vec{m}\rangle \right\rangle, \left\langle i; \vec{a}; \vec{t'}; \langle k; \vec{m}\rangle \right\rangle \in [\![\Gamma \vdash \mathcal{D}]\!]_{\gamma}$$

then, $\vec{t} = \vec{t'}$*.*

Proof. This immediately follows from the fact that for any two rules

$$\frac{X}{\langle i; \vec{a}; \vec{t}; \langle k; \vec{m}\rangle\rangle} \quad \text{and} \quad \frac{X'}{\left\langle i; \vec{a}; \vec{t'}; \langle k; \vec{m}\rangle\right\rangle}$$

in $\Phi^{\gamma}_{\Gamma,\mathcal{D}}$ we have $\vec{t} = \vec{t'}$. □

We shall show that the interpretation of the inductive types in a block are each in the universe corresponding to their arity. Notice that whenever two inductive types appear in one another the syntactic criteria for typing enforce that they are both have the same arity.[1] Therefore, we assume without loss of generality that all inductive types in the block have the same arity. In case it is not the case, it must be that there are some inductive types in the block that are not necessarily mutually inductive with the rest of the block. Hence, those inductive types (and their interpretations) can be considered prior to considering the block as a whole. Therefore, in the following theorem, we assume, without loss of generality that the all of the inductive types of a block are of the same arity.

Lemma B.3.8. *Let $\mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\}$ be a block of inductive types with* uniform *parameters such that all inductive types of arity* $\texttt{Type}_i$*. Furthermore, let us assume that all the terms (and particularly types) appearing in the body of the block are well defined under the context Γ and environment γ and interpretation of each of these terms (and types) is in the interpretation of the type (correspondingly sort) that is expected based on the typing derivation.*[2] *Then, $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma\downarrow$, $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma \in \mathcal{V}\kappa_j$, where* $\texttt{Type}_j$ *is the maximal* sort *of the inductive types in the block, and $[\![\Gamma \vdash \mathcal{D}.d_j]\!]_\gamma \in [\![\Gamma \vdash \Delta_I(d_j)]\!]_\gamma$.*

Proof. The construction of $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma$ depends only on the interpretation of terms $[\![\Gamma \vdash u]\!]_\gamma$ where u appears in Δ_I or Δ_C and by our assumptions these are all defined. Therefore, we can easily see that $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma\downarrow$. We show that $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma \in \mathcal{V}\kappa_j$. This follows by Lemma B.2.4. Notice that as all terms appearing in the inductive block have types that are in $\texttt{Type}_j$ we know that the cardinality of premises of rules in the rule set for constructing $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma$ are also all in $\mathcal{V}\kappa_j$. Since there are finitely many such terms we can take the maximum of these cardinalities which allows us to use Lemma B.2.4.

Let $\vec{b}$ be a sequence of sets that are in the interpretation for parameters of the mutual inductive block. Let $\mathcal{F}(\Phi) = \left\{a = \left\langle i; \vec{b}; \vec{t}; c\right\rangle \middle| \frac{A}{a} \in \Phi\right\}$ and $\mathcal{G}(A) = \left\{a = \left\langle i; \vec{b}; \vec{t}; c\right\rangle \middle| a \in A\right\}$. That is, $\mathcal{F}$ filters a rule set so that only

[1] Note that this is the case in our work as there are no inductive types in $\texttt{Prop}$.

[2] These conditions will hold by induction hypotheses when this lemma is used in practice.

those rules are retained that produce inductive types in the family indexed by parameter values $\vec{b}$. Similarly, $\mathcal{G}$ filters the fixpoint of such a rule set.

We show, by transfinite induction up to the closing ordinal of $\Phi^{\gamma}_{\Gamma,\mathcal{D}}$, that

$$\mathcal{G}(\mathcal{I}(\Phi^{\gamma}_{\Gamma,\mathcal{D}})) = \mathcal{I}(\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}}))$$

Notice that this crucially depends on the fact that parameters are uniform. Now, non-parameter arguments of constructors need to be in the sort (see the typing rule for inductive types). Therefore, for each fixed set of parameters, e.g., $\vec{b}$ above, for each constructor there is a fixed cardinality $\alpha \in \mathcal{V}\kappa_i$ (cardinality corresponding to the type of non-parameter arguments) such that the cardinality of premises of rules in $\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}})$ corresponding to that constructor is less than or to α. Since, there are finitely many such cardinalities we can take the maximum of these cardinalities which is again in $\mathcal{V}\kappa_i$. Hence, the closing ordinal of $\mathcal{I}(\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}}))$ is in $\mathcal{V}\kappa_i$. Notice, that this *does not* imply that $\mathcal{I}(\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}}))$ is in $\mathcal{V}\kappa_i$ as there are parameters can values for indices in the tuples in $\mathcal{I}(\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}}))$.

Finally, we show that for each sequence $\vec{c}$ of sets that are in the interpretation for the indices of an inductive type d_i, $\left\{a \middle| \left\langle i; \vec{b}; \vec{c}; a \right\rangle \mathcal{I}(\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}}))\right\} \in \mathcal{V}\kappa_i$. We show this by transfinite induction up to the closing ordinal of $\mathcal{I}(\mathcal{F}(\Phi^{\gamma}_{\Gamma,\mathcal{D}}))$ which, crucially, we know that is in $\mathcal{V}\kappa_i$. □

Lemma B.3.9. *Let $\mathcal{D} \equiv \mathsf{Ind}_n \{\Delta_I := \Delta_C\}$ and $\mathcal{D}' \equiv \mathsf{Ind}_n \{\Delta'_I := \Delta'_C\}$ be two blocks of inductive types with $\mathrm{dom}(\Delta_I) = \mathrm{dom}(\Delta'_I)$ and $\mathrm{dom}(\Delta_C) = \mathrm{dom}(\Delta'_C)$. Furthermore, assume that the for each $d \in \mathrm{dom}(\Delta_I)$ the interpretation of the arguments of the arity of $\Delta_C(d)$ are subsets of the interpretation of corresponding arguments of the arity of $\Delta_C(d')$. Similarly for the arguments of the constructors. In addition, assume that in each case, the interpretation of values given as parameters and arities in the resulting type of each corresponding constructors (the inductive type being constructed by that constructor) are equal. These are conditions in the rule* Ind-leq *in Figure 4.6 where cumulativity (subtyping) relation is replaced with subset relation on the interpretation and the judgemental equality is replaced with equality of interpretations.*[3]

Let $\vec{P}$ and $\vec{P'}$ be parameters of inductive blocks $\mathcal{D}$ and $\mathcal{D}'$, respectively. In this case,

$$\forall d \in \mathrm{dom}(\Delta_I).\ \Delta_I(d) \equiv \mathbf{\Pi} \vec{p} : \vec{P}.\, \vec{m} : \vec{M}.A_d \Rightarrow$$

$$\forall \vec{a}.\ \vec{a} \in \left[\!\left[\Gamma \vdash \vec{P}\right]\!\right]_{\gamma} \wedge \vec{a} \in \left[\!\left[\Gamma \vdash \vec{P'}\right]\!\right]_{\gamma} \Rightarrow$$

[3] These conditions will hold by the induction hypothesis when we shall use this lemma.

$$\forall \vec{b} \in \left[\!\left[\Gamma, \vec{p} : \vec{P} \vdash \vec{M}\right]\!\right]_\gamma \Rightarrow \overrightarrow{\mathsf{App}}([\![\Gamma \vdash \mathcal{D}.d]\!]_\gamma, \vec{a}, \vec{b}) \subseteq \overrightarrow{\mathsf{App}}([\![\Gamma \vdash \mathcal{D}'.d]\!]_\gamma, \vec{a}, \vec{b})$$

Proof. Expanding the definition of $[\![\Gamma \vdash \mathcal{D}.d]\!]_\gamma$ and $[\![\Gamma \vdash \mathcal{D}'.d]\!]_\gamma$ in the statement above gives us:

$$\forall d \in \mathrm{dom}(\Delta_I).\ \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{p} : \vec{P}.\vec{m} : \vec{M}.A_{d_i} \Rightarrow$$

$$\forall \vec{a}.\ \vec{a} \in \left[\!\left[\Gamma \vdash \vec{P}\right]\!\right]_\gamma \wedge \vec{a} \in \left[\!\left[\Gamma \vdash \vec{P'}\right]\!\right]_\gamma \Rightarrow$$

$$\forall \vec{b} \in \left[\!\left[\Gamma, \vec{p} : \vec{P} \vdash \vec{M}\right]\!\right]_\gamma \Rightarrow$$

$$\left\{c \middle| \left\langle i; \vec{a}; \vec{b}; c \right\rangle \in [\![\Gamma \vdash \mathcal{D}]\!]_\gamma\right\} \subseteq \left\{c \middle| \left\langle i; \vec{a}; \vec{b}; c \right\rangle \in [\![\Gamma \vdash \mathcal{D}']\!]_\gamma\right\}$$

In order to show this, we show, by transfinite induction on α up to the closing ordinal of $\Phi^\gamma_{\Gamma,\mathcal{D}}$, that the following holds

$$\forall d \in \mathrm{dom}(\Delta_I).\ \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{p} : \vec{P}.\vec{m} : \vec{M}.A_{d_i} \Rightarrow$$

$$\forall \vec{a}.\ \vec{a} \in \left[\!\left[\Gamma \vdash \vec{P}\right]\!\right]_\gamma \wedge \vec{a} \in \left[\!\left[\Gamma \vdash \vec{P'}\right]\!\right]_\gamma \Rightarrow$$

$$\forall \vec{b} \in \left[\!\left[\Gamma, \vec{p} : \vec{P} \vdash \vec{M}\right]\!\right]_\gamma \Rightarrow$$

$$\left\{c \middle| \left\langle i; \vec{a}; \vec{b}; c \right\rangle \in \mathcal{O}^\alpha_{\Phi^\gamma_{\Gamma,\mathcal{D}}}\right\} \subseteq \left\{c \middle| \left\langle i; \vec{a}; \vec{b}; c \right\rangle \in \mathcal{O}^\alpha_{\Phi^\gamma_{\Gamma,\mathcal{D}'}}\right\}$$

The base case, and the case for limit ordinals are trivial. In case of a successor ordinal, α^+, let a k be in $\left\{c \middle| \left\langle i; \vec{a}; \vec{b}; c \right\rangle \in \mathcal{O}^{\alpha^+}_{\Phi^\gamma_{\Gamma,\mathcal{D}}}\right\}$. Then it must be generated by a rule in $\Phi^\gamma_{\Gamma,\mathcal{D}}$. By the premise of this rule, we know that all the recursive (the same or other inductive types of the block) arguments are required to be in the previous stage, $\mathcal{O}^\alpha_{\Phi^\gamma_{\Gamma,\mathcal{D}}}$. By induction hypothesis, we know that those tuples should also belong to $\mathcal{O}^\alpha_{\Phi^\gamma_{\Gamma,\mathcal{D}'}}$. Hence, the corresponding rule exists in $\Phi^\gamma_{\Gamma,\mathcal{D}'}$ and is applicable. Therefore, the tuple k must also be in $\left\{c \middle| \left\langle i; \vec{a}; \vec{b}; c \right\rangle \in \mathcal{O}^{\alpha^+}_{\Phi^\gamma_{\Gamma,\mathcal{D}'}}\right\}$. □

B.3.2 Modeling eliminators

We define the interpretation for eliminators by constructing a rule set which will interpret the eliminators for the whole inductive block. We will define interpretation of individual eliminators based on the fixpoint of this rule set.

Definition B.3.10 (Interpretation of recursors)**.** *Let $\mathcal{D} \equiv \mathsf{Ind}_n \{\Delta_I := \Delta_C\}$ an arbitrary but fixed inductive block and assume, without loss of generality, that $\Delta_I = d_0 : A_0, \dots d_l : A_l$ and $\Delta_C = c_0 : T_0, \dots c_{l'} : T_{l'}$. Where $A_i \equiv \mathbf{\Pi} \vec{p} : \vec{P}. \mathbf{\Pi} \vec{x} : \vec{B}. s$ for some types $\vec{B}$ and some sort s. Here, $\vec{P}$ are parameters of the inductive block. The type of constructors are of the following form: $T_k \equiv \mathbf{\Pi} \vec{p} : \vec{P}. \mathbf{\Pi} \vec{x} : \overrightarrow{V_k}. d_{i_k}\ \vec{p}\ \vec{t_k}$ for some $\vec{t_k}$. For the sake of simplicity of presentation, let us write $ELB \equiv \mathsf{Elim}^{\mathcal{D}}(Q_{d_1}, \dots, Q_{d_l}) \{f_{c_1}, \dots, f_{c_{l'}}\}$ for the block of eliminators being interpreted. Furthermore, let $\xi_{\mathcal{D}}^{Q_{d_1},\dots,Q_{d_l}}(c_k, T_k) \equiv \mathbf{\Pi} \vec{x} : \overrightarrow{U_k}. Q_{d_{i_k}}\ \vec{u}$ for some terms $\vec{u}$. Notice that $\xi_{\mathcal{D}}^{Q_{d_1},\dots,Q_{d_l}}$ is simply the type of the constructor where after each (mutually) recursive argument, an argument is added for the result of the elimination of that (mutually) recursive argument. Let us write $J_{k,i}$ for the index of the i^{th} argument of the k^{th} constructor, $V_{k,i}$, in $\overrightarrow{U_k}$ above. More precisely, whenever $rec(V_{k,i})$ holds, we have $U_{J_{k,i}+1}$ is the argument of $\xi_{\mathcal{D}}^{Q_{d_1},\dots,Q_{d_l}}(c_k, T_k)$ that corresponds to the result of the elimination of $V_{k,i} = U_{J_{k,i}}$. We first define a rule set $\Phi^{\gamma}_{\Gamma,ELB}$ for this interpretation:*

$$\Phi^{\gamma}_{\Gamma,ELB} \triangleq \bigcup_{d_i \in \mathsf{dom}(\Delta_I)} \bigcup_{c_k \in \mathsf{Constrs}(d_i)} \left\{ \frac{\Psi_{d_i,c_k}}{\psi_{d_i,c_k}} \middle| \vec{a} \in \left[\!\left[\Gamma \vdash \overrightarrow{U_k} \right]\!\right]_{\gamma} \right\}$$

Let $\vec{b}$ be the subsequence of $\vec{a}$ corresponding to arguments of the constructor, i.e., it is obtained from $\vec{a}$ by dropping any term in the sequence that corresponds to some $U_{J_{k,j}+1}$ whenever $rec(V_{k,j})$.

$$\psi_{d_i,c_k} \triangleq \left\langle \left\langle c_k; \vec{b} \right\rangle ; \overrightarrow{\mathsf{App}}([\![\Gamma \vdash f_{c_k}]\!]_{\gamma}, \vec{a}) \right\rangle$$

For Ψ_k we assume that $V_{k,j} \equiv \mathbf{\Pi} \vec{y} : \overrightarrow{W_{k,j}}. d_{I_{k,j}}\ \vec{q}\ \vec{t}$.

$$\Gamma_{k,j} \triangleq \Gamma, \vec{p} : \vec{P}, x_1 : V_{k,1}, \dots, x_{j-1} : V_{k,j-1}$$

$$\vec{e} \triangleq b_1, \dots b_{j-1}$$

$$\Psi_{d_i,c_k} \triangleq \bigcup_{rec(V_{k,j})} \left\{ \left\langle \begin{array}{l} \overrightarrow{\mathsf{App}}([\![\Gamma_{k,j} \vdash U_{J_{k,j}}]\!]_{\gamma,\vec{e}}, \vec{u}); \\ \overrightarrow{\mathsf{App}}([\![\Gamma_{k,j} \vdash U_{J_{k,j}+1}]\!]_{\gamma,\vec{e}}, \vec{u}) \end{array} \right\rangle \middle| \vec{u} \in \left[\!\left[\Gamma_{k,j} \vdash \overrightarrow{W_{k,j}} \right]\!\right]_{\gamma,\vec{e}} \right\}$$

We define the interpretation individual eliminators as follows:

$$\left[\!\left[\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d_i; \vec{v}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}\right]\!\right]_\gamma \triangleq u$$

if $\left[\!\left[\Gamma \vdash t\right]\!\right]_\gamma = \langle k, \vec{m}\rangle$*, and* u *is the unique set such that* $\left\langle \left\langle k; \left[\!\left[\Gamma \vdash \vec{v}\right]\!\right]_\gamma, \vec{m}\right\rangle; u\right\rangle \in \mathcal{I}(\Phi^\gamma_{\Gamma, ELB})$.

The following lemma shows that in Definition B.3.10 above we have indeed captured and interpreted all of the elements of all of the inductive types in the block.

Lemma B.3.11. *Let* $\mathcal{D} \equiv \mathbf{Ind}_n\left\{\Delta_I := \Delta_C\right\}$ *be a block of inductive types with* $\left[\!\left[\Gamma \vdash \mathcal{D}\right]\!\right]_\gamma$ *defined and let* $Q_{d_1}, \ldots, Q_{d_l}$ *and* $f_{c_1}, \ldots, f_{c_{l'}}$ *be such that*

$$\left[\!\left[\Gamma \vdash Q_{d_i}\right]\!\right]_\gamma \in \left[\!\left[\Gamma \vdash \mathbf{\Pi}\vec{x} : \vec{A}.\, (d_i\ \vec{x}) \to s'\right]\!\right]_\gamma$$

and

$$\left[\!\left[\Gamma \vdash f_{c_i}\right]\!\right]_\gamma \in \left[\!\left[\Gamma \vdash \xi^{\vec{Q}}_{\mathcal{D}}(c_i, \Delta_C(c_i))\right]\!\right]_\gamma$$

Also, assume that for the term t *we have* $\left[\!\left[\Gamma \vdash t\right]\!\right]_\gamma \in \left[\!\left[\Gamma \vdash \mathcal{D}.d\ \vec{u}\ \vec{v}\right]\!\right]_\gamma$*. Then,*

$$\left[\!\left[\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}\right]\!\right]_\gamma\!\downarrow$$

and

$$\left[\!\left[\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d; \vec{u}; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}\right]\!\right]_\gamma \in \left[\!\left[\Gamma \vdash Q_{d_i}\ \vec{u}\ \vec{v}\ t\right]\!\right]_\gamma$$

Proof. We show by transfinite induction up to the closing ordinal of $\Phi^\gamma_{\Gamma,\mathcal{D}}$ that for any α and for any

$$\left\langle \langle k; \vec{a}, \vec{m}\rangle; \vec{t}\right\rangle \in \left\{ \left\langle \langle k; \vec{a}, \vec{m}\rangle; \vec{t}\right\rangle \middle| \langle i; \vec{a}; \vec{t}; \langle k; \vec{m}\rangle\rangle \in \mathcal{O}^\alpha_{\Phi^\gamma_{\Gamma,\mathcal{D}}} \right\}$$

(note that by Lemma B.3.7 $\vec{t}$ is uniquely determined by k, $\vec{a}$ and $\vec{m}$) there is a unique $b \in \left[\!\left[\Gamma \vdash \overrightarrow{\mathsf{App}}(Q_{d_i}, \vec{a}, \vec{t}, \langle k; \vec{m}\rangle)\right]\!\right]_\gamma$ such that $\langle \langle k; \vec{m}\rangle; b\rangle \in \mathcal{O}^\alpha_{\Phi^\gamma_{\Gamma,ELB}}$ for

$$ELB \equiv \mathbf{Elim}^{\mathcal{D}}(Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}$$

For the base case, $\alpha = 0$ this holds trivially. For the other cases, it suffices to notice that all the argument elements taken for the (mutually) recursive arguments of constructors, i.e., corresponding to $U_{J_{k,j}}$ for $rec(V_{J_{k,j}})$ are *uniquely* determined from the arguments of the constructor ($\vec{b}$ in the interpretation of recursors above) as is so restricted by the antecedents of each rule.

Notice that the argument above also shows that $\Phi^\Gamma_{\Gamma,ELB}$ and $\Phi^\Gamma_{\Gamma,\mathcal{D}}$ have the same closing ordinal. This concludes the proof. □

B.3.3 Proof of Soundness

We show that the model that we have constructed throughout this section is sound. That is, we show that for any typing context Γ, term t and type A such that $\Gamma \vdash t : A$ we have that for any environment $\gamma \in [\![\Gamma]\!]$, $[\![\Gamma \vdash t]\!]_\gamma\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\downarrow$ and that $[\![\Gamma \vdash t]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$. We use this result to prove consistency of PCUIC.

Lemma B.3.12. *Let Γ be a typing context, γ be an environment, t such that $[\![\Gamma \vdash t]\!]_\gamma\downarrow$. Then $FV(t) \subseteq \mathsf{dom}(\Gamma)$.*

Proof. We prove that if there is a variable $x \in FV(t)$ such that $x \notin \mathsf{dom}(\Gamma)$ then $\neg\, [\![\Gamma \vdash t]\!]_\gamma\downarrow$. This follows easily by induction on t. □

Lemma B.3.13 (Weakening)**.** *Let Γ be a typing context, γ be an environment, t and A be terms such that $[\![\Gamma, \Gamma']\!]\downarrow$, $\gamma \in [\![\Gamma]\!]$, $\gamma, \gamma' \in [\![\Gamma, \Gamma']\!]$ and $[\![\Gamma, \Gamma' \vdash t]\!]_{\gamma,\gamma'}\downarrow$. Furthermore, let Ξ be a typing context and δ be an environment such that $(\mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Gamma')) \cap \mathsf{dom}(\Xi) = \emptyset$, and we have that variables in $\mathsf{dom}(\Xi)$ do not appear in Γ' freely such that $[\![\Gamma, \Xi, \Gamma']\!]\downarrow$ and $\gamma, \delta, \gamma' \in [\![\Gamma, \Xi, \Gamma']\!]$. Then*

1. $[\![\Gamma, \Xi, \Gamma' \vdash t]\!]_{\gamma,\delta,\gamma'}\downarrow$
2. $[\![\Gamma, \Xi, \Gamma' \vdash t]\!]_{\gamma,\delta,\gamma'} = [\![\Gamma, \Gamma' \vdash t]\!]_{\gamma,\gamma'}$

Proof. We prove the result above by induction on t. Most cases follow easily by induction hypotheses. Here, for demonstration purposes we show the case for variables and (dependent) function types.

- $t = x$: In this case, both Case 1 and Case 2 above follow by definition of the model.
- $t = \mathbf{\Pi} x : A.\, B$: We know by induction hypothesis that

$$[\![\Gamma, \Xi, \Gamma' \vdash A]\!]_{\gamma,\delta,\gamma'}\downarrow$$

$$[\![\Gamma, \Xi, \Gamma' \vdash A]\!]_{\gamma,\delta,\gamma'} = [\![\Gamma, \Gamma' \vdash A]\!]_{\gamma,\gamma'}$$

  Also by induction hypotheses we have that for all $a \in [\![\Gamma, \Xi, \Gamma' \vdash A]\!]_{\gamma,\delta,\gamma'}$,

$$([\![\Gamma, \Xi, \Gamma', x : A \vdash B]\!]_{\gamma,\delta,\gamma',a})\downarrow$$

$$[\![\Gamma, \Xi, \Gamma', x : A \vdash B]\!]_{\gamma,\delta,\gamma',a} = [\![\Gamma, \Gamma', x : A \vdash B]\!]_{\gamma,\gamma',a}$$

  Given the above induction hypotheses, both Case 1 and Case 2 above follow by the definition of the model. Note that by Lemma B.3.12, we know that variables in $\mathsf{dom}(\Xi)$ do not appear freely in A. This is why induction hypothesis applies to B above. □

Lemma B.3.14 (Substitutivity)**.** *Let Γ be a typing context, γ be an environment, u and A be terms such that $[\![\Gamma]\!]\downarrow$, $\gamma \in [\![\Gamma]\!]$ and $[\![\Gamma \vdash u]\!]_\gamma\downarrow$. Then*

1. *If $[\![\Gamma, x : A, \Xi]\!]\downarrow$ and $\gamma, [\![\Gamma \vdash u]\!]_\gamma, \delta \in [\![\Gamma, x : A, \Xi]\!]$ then $\gamma, \delta \in [\![\Gamma, \Xi[u/x]]\!]$*
2. *If*
$$[\![\Gamma, x : A, \Xi]\!]\downarrow \quad \text{and} \quad \gamma, [\![\Gamma \vdash u]\!]_\gamma, \delta \in [\![\Gamma, x : A, \Xi]\!] \quad \text{and}$$
$$[\![\Gamma, x : A, \Xi \vdash t]\!]_{\gamma, [\![\Gamma \vdash u]\!]_\gamma, \delta}\downarrow$$
then
   (a) $[\![\Gamma, \Xi[x/u] \vdash t[u/x]]\!]_{\gamma,\delta}\downarrow$
   (b) $[\![\Gamma, \Xi[x/u] \vdash t[u/x]]\!]_{\gamma,\delta} =$
   $[\![\Gamma, x : A, \Xi \vdash t]\!]_{\gamma, [\![\Gamma \vdash u]\!]_\gamma, \delta} = [\![\Gamma, x := u : A, \Xi \vdash t]\!]_{\gamma, [\![\Gamma \vdash u]\!]_\gamma, \delta}$

Proof. We prove this lemma by well-founded induction on $\sigma(\Xi, t) = \mathit{size}(\Xi) + \mathit{size}(t) - 1/2$. In the following we reason as though we are conducting induction on the structure of terms and contexts. Note that this is allowed because our measure $\sigma(\Xi, t)$ does decrease for the induction hypotheses pertaining to structural induction in each case $\sigma(\Xi, t)$ is decreasing. This is in particular crucial in some sub-cases of the case where $\Xi = \Xi, y : B$. There, we are performing induction on t which enlarges the context Ξ (similar to case of (dependent) functions in Lemma B.3.13).

- Case $\Xi = \cdot$: then Case 1 holds trivially. Case 2, follows by induction on t. The only non-trivial case (not immediately following from the induction hypothesis) is the case where t is a variable. In this case, both 2a and 2b follow by definition of the model.
- Case $\Xi = \Xi', y : B$ and correspondingly, $\delta = \delta', a$:
   1. Follows by definition of the model and the induction hypothesis corresponding to Case 2b:
   $$[\![\Gamma, \Xi'[u/x] \vdash B[u/x]]\!]_{\gamma,\delta'} = [\![\Gamma, x : A, \Xi' \vdash B]\!]_{\gamma, [\![\Gamma \vdash u]\!]_\gamma, \delta'}$$
   2. We proceed by induction on t. The only non-trivial case is when t is a variable, $t = z$. Notice, when $z \neq x$ then both 2a and 2b hold trivially by definition of the model. Otherwise, we have to show that $[\![\Gamma, \Xi'[u/x], y : B[u/x] \vdash u]\!]_{\gamma,\delta,a}\downarrow$ and
   $$[\![\Gamma, \Xi[u/x], y : B[u/x] \vdash u]\!]_{\gamma,\delta,a} =$$

$$[\![\Gamma, x : A, \Xi, y : B[u/x] \vdash x]\!]_{\gamma,[\![\Gamma\vdash u]\!]_\gamma,\delta.a} =$$

$$[\![\Gamma, x := u : A, \Xi, y : B[u/x] \vdash x]\!]_{\gamma,[\![\Gamma\vdash u]\!]_\gamma,\delta,a}$$

These follow by Lemma B.3.13 above and our assumption that $[\![\Gamma \vdash u]\!]_\gamma\downarrow$.

- Case $\Xi = \Xi', \mathbf{Ind}_n\{\Delta_I := \Delta_C\}$: We prove this case by induction on t. The only non-trivial case here is the case where $t = \mathbf{Ind}_n\{\Delta_I := \Delta_C\}.z$. Notice that the interpretation of $[\![\Gamma, \Xi'[u/x], y : B[u/x] \vdash t]\!]_{\gamma,\delta,a}$ is defined based on the interpretation of terms of the form $[\![\Gamma, \Xi'[u/x] \vdash m]\!]_{\gamma,\delta}$ where m appears in Δ_I or Δ_C to which the induction hypothesis applies. □

Proof of soundness of the model (Lemma 4.5.1). Note that judgements of the form $\mathcal{WF}(\Gamma)$, $\Gamma \vdash t : A$, $\Gamma \vdash t \simeq t' : A$ and $\Gamma \vdash A \preceq B$ are defined mutually. We prove the theorem by mutual induction on the derivation of these judgements.

- Case WF-CTX-EMPTY: trivial by definition.
- Case WF-CTX-HYP: trivial by definition and induction hypothesis.
- Case WF-CTX-DEF: trivial by definition and induction hypothesis.
- Case PROP: trivial by definition.
- Case HIERARCHY: trivial by definition.
- Case VAR: trivial by definition and induction hypotheses.
- Case LET: by definition, induction hypothesis and Lemma B.3.14.
- Case LET-EQ: by definition, induction hypotheses and Lemma B.3.14.
- Case PROD: by induction hypotheses we know that $[\![\Gamma \vdash A]\!]_\gamma \in [\![\Gamma \vdash s_1]\!]_\gamma$ and that $[\![\Gamma, x : A \vdash B]\!]_{\gamma,a} \in [\![\Gamma \vdash s_2]\!]_\gamma$ for any $a \in [\![\Gamma \vdash A]\!]_\gamma$. We also know $\mathcal{R}_s(s_1, s_2, s_3)$. We have to show that

$$[\![\Gamma \vdash \mathbf{\Pi} x : A.\, B]\!]_\gamma \in [\![\Gamma \vdash s_3]\!]_\gamma \tag{B.1}$$

Since $\mathcal{R}_s(s_1, s_2, s_3)$ holds, we have that either $s_2 = s_3 = \texttt{Prop}$ or $s_1 = \texttt{Type}_i$, $s_2 = \texttt{Type}_j$ and $s_3 = \texttt{Type}_{\max\{i,j\}}$ or $s_1 = \texttt{Prop}$, $s_2 = \texttt{Type}_i$ and $s_3 = \texttt{Type}_i$ hold. In the first case, The membership relation above, (B.1), follows from Case 1 of Lemma B.2.8. In the other two cases, note that $[\![\Gamma \vdash s_3]\!]_\gamma$ is a von Neumann universe and is hence closed under (dependent) function space and also (by axiom schema of replacement) under taking trace-encoding of elements of any set.

- Case Prod-eq: by definition and induction hypotheses.
- Case Lam: by definition and induction hypotheses.
- Case Lam-eq: by definition and induction hypotheses.
- Case App: by induction hypotheses we know that $[\![\Gamma \vdash N]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$ and $[\![\Gamma \vdash M]\!]_\gamma \in [\![\Gamma \vdash \mathbf{\Pi} x : A.\, B]\!]_\gamma$. From the latter we know that $[\![\Gamma \vdash M]\!]_\gamma = \mathsf{Lam}(f)$ for some f with domain $\mathsf{dom}(f) = [\![\Gamma \vdash A]\!]_\gamma$ such that $f(a) = [\![\Gamma, x : A \vdash B]\!]_{\gamma,a}$. Therefore,

  $$[\![\Gamma \vdash M\ N]\!]_\gamma = \mathsf{App}(\mathsf{Lam}(f), [\![\Gamma \vdash N]\!]_\gamma)$$
  $$= f([\![\Gamma \vdash N]\!]_\gamma) \in [\![\Gamma, x : A \vdash B]\!]_{\gamma, [\![\Gamma \vdash N]\!]_\gamma} = [\![\Gamma \vdash B[N/x]]\!]_\gamma$$

  where the last equality is by Lemma B.3.14.
- Case App-eq: by a reasoning similar to that of Case App above.
- Case Ind-WF: by definition and induction hypotheses and Lemma B.3.8.
- Case Ind-type: by induction hypotheses we know that $[\![\Gamma]\!]\downarrow$ which by definition means that for any environment $\gamma \in [\![\Gamma]\!]$ and any inductive block $\mathcal{D} \in \Gamma$ we have (using the weakening lemma above) $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma\downarrow$. By Lemma B.3.8 the desired result follows.
- Case Ind-constr: by induction hypotheses we know that $[\![\Gamma]\!]\downarrow$ which by definition means that for any environment $\gamma \in [\![\Gamma]\!]$ and any inductive block $\mathcal{D} \in \Gamma$ we have (using the weakening lemma above) $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma\downarrow$. The desired result follows by the definition of $[\![\Gamma \vdash \mathcal{D}.c]\!]_\gamma$, Lemma B.3.14 and the fact that $[\![\Gamma \vdash \mathcal{D}]\!]_\gamma\downarrow$ is the fixpoint of the rule-set used to construct it.
- Case Ind-Elim: by Lemma B.3.11 and induction hypotheses.
- Case Ind-Elim-eq: similarly to Case Ind-Elim.
- Case Eq-ref: trivial by definition and induction hypothesis.
- Case Eq-sym: trivial by definition and induction hypothesis.
- Case Eq-trans: trivial by definition and induction hypothesis.
- Case Beta: by induction hypotheses that

  $$[\![\Gamma, x : A \vdash M]\!]_\gamma\downarrow$$
  $$\forall a \in [\![\Gamma \vdash A]\!]_\gamma\,.\ [\![\Gamma, x : A \vdash M]\!]_{\gamma,a} \in [\![\Gamma, x : A \vdash B]\!]_{\gamma,a}$$

  On the other hand, we have that

$$\llbracket \Gamma \vdash N \rrbracket_\gamma \downarrow$$
$$\llbracket \Gamma \vdash N \rrbracket_\gamma \in \llbracket \Gamma \vdash A \rrbracket_\gamma$$
$$\llbracket \Gamma \vdash A \rrbracket_\gamma \downarrow$$

and therefore

$$\llbracket \Gamma, x : A \vdash B \rrbracket_{\gamma, \llbracket \Gamma \vdash N \rrbracket_\gamma} \in \llbracket \Gamma \vdash s \rrbracket_\gamma$$

Also, by Lemma B.3.14 we get that

$$\llbracket \Gamma \vdash B[N/x] \rrbracket_\gamma = \llbracket \Gamma, x : A \vdash B \rrbracket_{\gamma, \llbracket \Gamma \vdash N \rrbracket_\gamma} \in \llbracket \Gamma \vdash s \rrbracket_\gamma$$

and that

$$\llbracket \Gamma \vdash M[N/x] \rrbracket_\gamma = \llbracket \Gamma, x : A \vdash M \rrbracket_{\gamma, \llbracket \Gamma \vdash u \rrbracket_\gamma} \in \llbracket \Gamma, x : A \vdash B \rrbracket_{\gamma, \llbracket \Gamma \vdash u \rrbracket_\gamma}$$

We finish the proof by:

$$\begin{aligned}
&\llbracket \Gamma \vdash (\lambda x : A.\, M)\ N \rrbracket_\gamma \\
&= \mathsf{App}(\llbracket \Gamma \vdash \lambda x : A.\, M \rrbracket_\gamma, \llbracket \Gamma \vdash N \rrbracket_\gamma) \\
&= \mathsf{App}(\mathsf{Lam}(a \in \llbracket \Gamma \vdash A \rrbracket_\gamma \mapsto \llbracket \Gamma, x : A \vdash M \rrbracket_{\gamma,a}), \llbracket \Gamma \vdash N \rrbracket_\gamma) \\
&= \llbracket \Gamma, x : A \vdash M \rrbracket_{\gamma, \llbracket \Gamma \vdash N \rrbracket_\gamma} \\
&= \llbracket \Gamma \vdash M[N/x] \rrbracket_\gamma
\end{aligned}$$

- Case Eta: We need to show that

$$\llbracket \Gamma \vdash t \rrbracket_\gamma = \llbracket \Gamma \vdash \lambda x : A.\, t\ x \rrbracket_\gamma \in \llbracket \Gamma \vdash \mathbf{\Pi} x : A.\, B \rrbracket_\gamma$$

We know by induction hypothesis that $\llbracket \Gamma \vdash t \rrbracket_\gamma \downarrow$ and that $\llbracket \Gamma \vdash t \rrbracket_\gamma \in \llbracket \Gamma \vdash \mathbf{\Pi} x : A.\, B \rrbracket_\gamma$ and consequently we know that there is a set theoretic function f:

$$f \in \Pi a \in \llbracket \Gamma \vdash A \rrbracket_\gamma.\ \llbracket \Gamma, x : A \vdash B \rrbracket_{\gamma,a}$$

such that $\llbracket \Gamma \vdash t \rrbracket_\gamma = \mathsf{Lam}(f)$. This implies that,

$$\begin{aligned}
\llbracket \Gamma \vdash \lambda x : A.\, t\ x \rrbracket_\gamma &= \mathsf{Lam}\left(\left\{(a, \llbracket \Gamma, x : A \vdash t\ x \rrbracket_{\gamma,a}) \middle| a \in \llbracket \Gamma \vdash A \rrbracket_\gamma \right\}\right) \\
&= \mathsf{Lam}\left(\left\{(a, \mathsf{App}(\llbracket \Gamma, x : A \vdash t \rrbracket_{\gamma,a}, a)) \middle| a \in \llbracket \Gamma \vdash A \rrbracket_\gamma \right\}\right) \\
&= \mathsf{Lam}\left(\left\{(a, \mathsf{App}(\llbracket \Gamma \vdash t \rrbracket_{\gamma,a}, a)) \middle| a \in \llbracket \Gamma \vdash A \rrbracket_\gamma \right\}\right)
\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{Lam}\left(\left\{(a, \mathsf{App}(\mathsf{Lam}(f), a)) \middle| a \in [\![\Gamma \vdash A]\!]_\gamma\right\}\right) \\
&= \mathsf{Lam}\left(\left\{(a, f(a)) \middle| a \in [\![\Gamma \vdash A]\!]_\gamma\right\}\right) \\
&= \mathsf{Lam}(f) \\
&= [\![\Gamma \vdash t]\!]_\gamma
\end{aligned}
$$

- Case Delta: by Lemma B.3.14 and induction hypotheses.
- Case Zeta: trivial by definition and induction hypothesis.
- Case Iota: by definition of the interpretation of recursors and induction hypothesis. Notice that by construction of the interpretation of eliminators in Definition B.3.10 the interpretation of elimination of $c_i\ \vec{a}$ is basically, the result of applying $f_{c_i}\ \vec{b}$ where $\vec{a}$ is a subsequence of $\vec{b}$. The only difference between $\vec{b}$ and $\vec{a}$ is that in $\vec{b}$ after each value that corresponds to a (mutually) recursive argument of the constructor c_i we have a value that corresponds to the interpretation of elimination of that (mutually) recursive argument. For those terms, $\mu_{\mathcal{D}}^{\vec{Q};\vec{f}}$ applies the elimination using the eliminator, **Elim**, while the rule in rule set corresponding $c_i\ \vec{a}$ ensures in its antecedents that all elements are eliminated correctly according to the interpretation of the eliminator (the fixpoint taken in Definition B.3.10). Notice that if the constructor is of an inductive sub-type, then, by construction of interpretation of constructors the interpretation of the two constructors applied to those terms are *equal*.
- Case Prop-in-Type: trivial by definition.
- Case Cum-Type: trivial by definition.
- Case Cum-trans: trivial by definition and induction hypothesis.
- Case Cum-weaken: trivial by definition, induction hypothesis and Lemma B.3.13.
- Case Cum-Prod: trivial by definition and induction hypothesis.
- Case Cum-eq-L: trivial by definition and induction hypothesis.
- Case Cum-eq-R: trivial by definition and induction hypothesis.
- Case Cum: trivial by definition and induction hypothesis.
- Case Cum-eq: trivial by definition and induction hypothesis.

- Case Eq-Cum: trivial by definition and induction hypothesis.
- Case C-Ind: easy by definition, induction hypothesis and Lemma B.3.9.
- Case Ind-Eq: by definition, induction hypotheses and Lemma B.3.9.
- Case Constr-Eq-L: by definition, induction hypothesis.
- Case Constr-Eq-R: by definition, induction hypothesis. □

Appendix C

Iris resources

Resources in Iris are described using a kind of partial commutative monoids, and the user of the logic can introduce new monoids.[1] For instance, in the case of finite partial maps, the partiality comes from the fact that disjoint union of finite maps is partial. Undefinedness is treated by means of a validity predicate $\checkmark : \mathcal{M} \to \mathit{iProp}$, which expresses which elements of the monoid $\mathcal{M}$ are valid/defined.

We write $\boxed{a : \mathcal{M}}^{\gamma}$ to assert that a monoid instance named γ, of type $\mathcal{M}$, has contents a. Often, we disregard the type if it is obvious from the context and simply write $\boxed{a}^{\gamma}$. We think of this assertion as a ghost variable γ with contents a.

GHOST-ALLOC
$\checkmark a \vdash \Rightarrow\!\!\!\mid \exists \gamma.\, \boxed{a}^{\gamma}$

OWN-VALID
$\boxed{a}^{\gamma} \vdash \checkmark(a)$

SHARING
$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$

Some Useful Monoids Here, we describe a few monoids which are particularly useful and which we will use in the sequel. We do not give the full definitions of the monoids (those can be found in (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017)), but focus instead on the properties which the elements of the monoids satisfy, shown in Figure C.1. These rules stated are only for monoids that we use in this work and not in Iris in its generality. For instance,

[1] Technically these are resource algebras (RAs) which are similar to monoids. In particular, RAs need not necessarily have a unit element. Furthermore, RAs are step-indexed. We ignore these details here as they are not directly relevant to our discussions. For more detail see Jung, Krebbers, Birkedal, and Dreyer (2016) and Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal (2017).

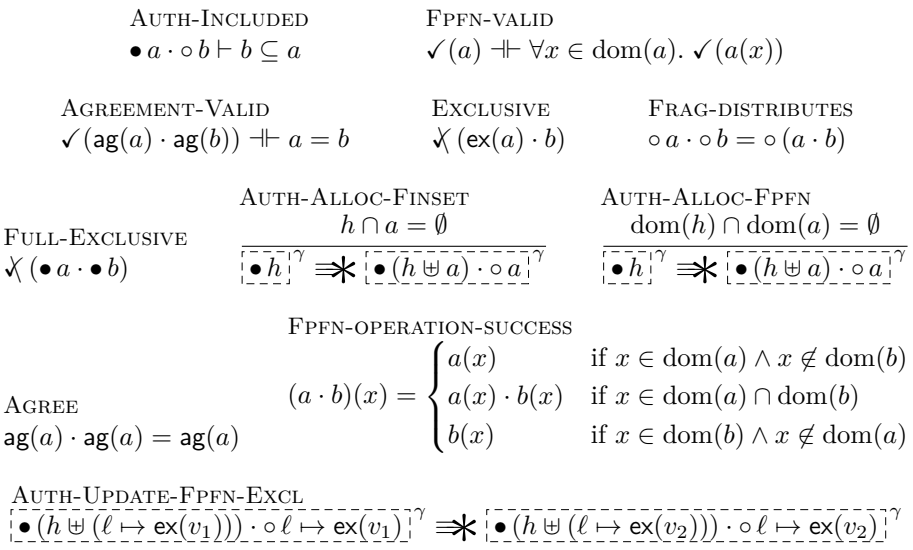


Figure C.1: Rules for selected monoid resources in Iris

in the rule Auth-Included, $\subseteq$ is a set relation and is defined for finite set and finite partial function monoids and not in general.

Figure C.1 depicts the rules necessary for allocating and updating finite set monoids, finset(A), and finite partial function monoids, $A \rightharpoonup^{\mathrm{fin}} M$. For finset$(A)$, the monoid operation $x \cdot y$ is union. The notation $a \mapsto b : A \rightharpoonup^{\mathrm{fin}} B \triangleq \{(a, b)\}$ is a singleton finite partial function.

The constructs $\bullet$ and $\circ$ are constructors of the so-called authoritative monoid Auth(M). We read $\bullet a$ as "*full* a" and $\circ a$ as "*fragment* a". We use the authoritative monoid to distribute ownership of fragments of a resource. The intuition is that $\bullet a$ is the authoritative knowledge of the full resource, think of it as being kept track of in a central location (see rule Auth-Included). The fragments, $\circ a$, can be shared (rule Frag-distributes) while the full part (the central location) should always remain unique (rule Full-Exclusive).

In addition to authoritative monoids, we also use the agreement monoid Ag(M) and exclusive monoid Ex(M). As the name suggests, the operation of the agreement monoid guarantees that $\mathsf{ag}(a) \cdot \mathsf{ag}(b)$ is invalid whenever $a \neq b$ (and otherwise it is idempotent; see rules Agree and Agreement-Valid). From the

rule AGREE it follows that the ownership of elements of $\text{AG}(M)$ is duplicable.[2]

$$\boxed{\mathsf{ag}(a)}^{\gamma} \dashv\vdash \boxed{\mathsf{ag}(a) \cdot \mathsf{ag}(a)}^{\gamma} \dashv\vdash \boxed{\mathsf{ag}(a)}^{\gamma} * \boxed{\mathsf{ag}(a)}^{\gamma}$$

The operation of the exclusive monoid never results in a valid element (rule EXCLUSIVE), enforcing that there can only be one instance of it owned. We can define the resource for keeping track of the physical heap of the programming language, HEAP. This is the canonical example of a monoid.

$$\text{HEAP} \triangleq \text{AUTH}(\mathit{Loc} \xrightarrow{\text{fin}} (\text{EX}(\mathit{Val})))$$

Hence, the points-to proposition of the separation logic can be defined as follows.

$$\ell \mapsto v \triangleq \boxed{\circ [l \mapsto \mathsf{ex}(v)]}^{\gamma_h}$$

Here, γ_h is the *globally fixed* monoid name to keep track of the heap of $\mathsf{F}_{\mu,ref,conc}$. The full part of this monoid is used in the definition of the weakest precondition to allow weakest preconditions to refer to the physical state of the program (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017). Notice here that HEAP is build from nesting EX in the finite partial functions monoid, which again is nested in the AUTH monoid. Therefore, to allocate and update the HEAP monoid, we can use AUTH-ALLOC-FPFN and AUTH-UPDATE-FPFN-EXCL respectively.

[2] Indeed it can be shown that elements of $\text{AG}(M)$ are also persistent.

Appendix D

Details of type soundness and refinements in Iris

D.1 Proof of semantic well-typedness of a program that is not syntactically well-typed

The program below is not syntactically of the type well-typed. However, it is semantically of the type $\forall X.\, \forall Y.\, (\texttt{ref}(X) \to X) \to X \to (\texttt{ref}(Y) \to Y) \to Y \to X + Y$.

$$\Lambda\,\Lambda\,(\lambda f.\, \lambda x.\, \lambda g.\, \lambda y.\, \texttt{let}\, l = \texttt{ref}(\texttt{true})\, \texttt{in}\, \texttt{fork}\, \{l \leftarrow \texttt{false}\}\,;$$
$$\texttt{if}\, !l\, \texttt{then}\, \mathit{waitfor}\; l; l \leftarrow x; \texttt{inj}_1\,(f\; l)\, \texttt{else}\, l \leftarrow y; \texttt{inj}_2\,(g\; l))$$

where

$$\mathit{waitfor} \triangleq \texttt{rec}\, f(x) = \texttt{if}\, !x\, \texttt{then}\, f\; x\, \texttt{else}\, ()$$

To prove this we use an invariant randInv:

$$\text{randInv}(\ell) \triangleq \boxed{\ell \mapsto \texttt{true} \vee (\ell \mapsto \texttt{false} * \mathit{tok}_1) \vee (\mathit{tok}_1 * \mathit{tok}_2)}^{\mathcal{N}.rnd}$$

Propositions tok_1 and tok_2 are exclusive tokens, i.e., two instances (with different names) of the monoid Ex(1). The idea is that the protocol regarding the sharing of location l (the result of allocation of `true`) can be in one of the following three states: (1) ℓ is just allocated and has not been set to `false` (the left disjunct in the invariant), (2) the other thread has performed writing `false`

(the middle disjunct in the invariant), and (3) the main thread has noticed that the other thread has finished (the right disjunct in the invariant). We start out by allocating the two tokens when we allocate `true`. Subsequently, we establish the invariant retaining both tokens. We give the invariant and the token tok_1 to the launched thread. This thread can use exclusivity of tok_1 to ensure that the protocol is in stage 1 before writing the value `false`. Furthermore, it has to give this token up when it succeeds in writing the value `false` to be able to close the invariant back. The main thread retaining the token tok_2 knows, when reading the reference, and subsequently when waiting (if it has to), that the protocol is not over. It has to give up tok_2 to be able to take out the points-to proposition out of the invariant at the end of waiting (or reading `false` the first time). At the end, the main thread will have $\ell \mapsto$ `false` before it has to overwrite it with the appropriate value and establish the semantic typing of the reference ℓ.

D.2 Monoids for evaluation on the specification side

In this section we discuss monoids that we use for reasoning about the evaluation on the specification side. For the heap of the specification side we use (another instance of) the exact same monoid Heap that we used for the heap of the implementation side above in Appendix C. For the thread pool of the specification side we use the monoid Tpool:[1]

$$\textsc{Tpool} \triangleq \textsc{Auth}(\mathbb{N} \xrightarrow{\text{fin}} (\textsc{Ex}(\mathit{Expr})))$$

We use *globally fixed* monoid names γ'_h and γ'_{tp} to respectively keep track of the specification side heap and thread pool. Given these, we define SpecConf, $\Mapsto$ and $\mapsto_s$ as follows:

$$\text{SpecConf}(\sigma, \vec{e}) \triangleq \boxed{\bullet \text{res}(\sigma)}^{\gamma'_h} * \boxed{\bullet \text{fpfnOf}(\vec{e})}^{\gamma'_{tp}}$$

$$\ell \mapsto_s v \triangleq \boxed{\circ [\ell \mapsto v]}^{\gamma'_h}$$

$$j \Mapsto_s e \triangleq \boxed{\circ [j \mapsto e]}^{\gamma'_{tp}}$$

where

$$\text{fpfnOf}(e_1, \ldots, e_n) \triangleq \{(i, e_i) | 1 \leq i \leq n\}$$

[1] In the actual Coq implementation we use the product of these two monoids to represent configurations as a single monoid. However, it is conceptually the same, and easier to explain on paper, when two separate monoids are considered.

Using the definitions above and the rules for allocating and updating resources in Figure C.1 we can easily derive all the necessary rules in Section 5.5.

D.3 The spin lock implementation

The compare and set operation (`CAS`) of $\mathsf{F}_{\mu,ref,conc}$ can be used to construct a spin lock. The following piece of code implements a spin lock as a boolean reference that is set to `false` whenever the lock is free. The release operation simply sets the lock to `false`. The acquire operation, on the other hand, tries to acquire the lock by attempting to atomically set it to `true` if it is `false`. In case it fails, i.e., the lock is acquired by another thread, it tries again. In other words, the acquire operation busy-waits until the lock is free and it can acquire it.

```
let new_lock () = let l = ref false

let acquire l =
  let rec try_acquire () =
  if CAS(l, false, true) then
    ()
  else
    try_acquire ()

let release l = l := false
```

D.4 Monoids for contextual refinement of stacks

In this part we describe how the predicates AllCells and $\mapsto^{stk}$ are defined. For this purpose we use a globally fixed instance, named γ_{st}, of the monoid STK defined below.

$$\text{STK} \triangleq \textsc{Auth}(\ell \xrightarrow{\text{fin}} (\textsc{Ag}(v)))$$

Note the similarity between this monoid and the monoid Heap representing heaps. The only difference is that STK is defined using the agreement monoid while Heap is defined using the exclusive monoid. We will discuss the consequences of this difference below.

We define the predicates AllCells and $\mapsto^{stk}$ as follows:

$$\text{AllCells}(f) \triangleq \boxed{\bullet\, \text{cellsOf}(f)}^{\gamma_{stk}}$$

$$\ell \mapsto^{stk} v \triangleq \boxed{\circ[\ell \mapsto v]}^{\gamma_{stk}}$$

where

$$\text{cellsOf}(f) \triangleq \{(\ell, \mathsf{ag}(v)) | (\ell, v) \in f\}$$

The definitions above along with the rules for allocating and updating resources in Figure C.1 allow us to easily derive all the necessary rules in Section 5.7. In particular, due to the properties of the agreement monoid we have that $\ell \mapsto^{stk} v$ is persistent.

Appendix E

Details of Logical Relations for Monadic Encapsulation of State and its Coq Formalization

E.1 Iris Definitions of Predicates used in the Logical Relation

In this section we detail how the abstract predicates ($\mathsf{regions}$, $\mathsf{region}(r, \gamma_h, \gamma'_h)$, $\mathsf{isRgn}(\alpha, r)$, $heap_{\gamma_h}(h)$ and $\ell \mapsto_\gamma v$) used in the definition of the logical relation are precisely defined in the Iris logic. To this end, we first introduce three more concepts from the Iris logic: invariants, saved predicates and ghost-state.

E.1.1 Saved Predicates

For storing of Iris propositions we use a mechanism called saved predicates, $\gamma \Rightarrow \Phi$. This is simply a convenient way of assigning a name γ to a predicate Φ. There are only three rules governing the use of saved predicates. We can allocate them (rule SavedPred-Alloc), they are persistent (rule SavedPred-Persistent) and the association of names to predicates is functional (rule

SavedPred-Equiv).

$$\textsc{SavedPred-Alloc}\quad \Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{E}} \exists\gamma.\ \gamma \Rightarrow \Phi \qquad \textsc{SavedPred-Persistent}\quad \gamma \Rightarrow \Phi \dashv\vdash \gamma \Rightarrow \Phi * \gamma \Rightarrow \Phi \qquad \frac{\textsc{SavedPred-Equiv}\quad \gamma \Rightarrow \Phi * \gamma \Rightarrow \Psi}{\triangleright \Phi(a) \vdash \triangleright \Psi(a)}$$

The later modality is used in rule SavedPred-Equiv as a guard to avoid self referential paradoxes Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017, which is not so surprising, after all, since saved predicates essentially allow us to store a predicate (something of type $\kappa \to \mathit{iProp}$) inside a proposition (something of type iProp).

Resources for the heap of STLang We use the monoid Heap defined in Appendix C to model the heap of the STLang. We define the propositions $\mathit{heap}_\gamma(h)$ and $\ell \mapsto_\gamma v$ as follows:

$$\mathit{heap}_\gamma(h) \triangleq \boxed{\bullet h}^{\gamma} \qquad \ell \mapsto_\gamma v \triangleq \boxed{\circ [l \mapsto \mathsf{ex}(v)]}^{\gamma}$$

Notice here that Heap is build from nesting Ex in the finite partial functions monoid, which again is nested in the Auth monoid. Therefore, to allocate and update and in the Heap monoid, we can use Auth-Alloc-Fpfn and Auth-Update-Fpfn-Excl respectively.

E.1.2 Encoding of Regions by Ghost Resources

Remark E.1.1. *In this section we will use Iris invariants. However, for the sake of simplicity, will ignore the names of invariants and the masks of the update modalities. For more details about invariants see Section 5.3.*

In order to concretely represent bijections and relatedness between locations, we use a pair of monoids, one for the bijection (one-to-one correspondence) and one for the semantic interpretation, i.e., a name to a saved predicate:

$$\mathsf{Rel} \triangleq \textsc{Auth}((\mathit{Loc} \times \mathit{Loc}) \xrightarrow{\text{fin}} (\textsc{Ag}(\mathit{Names}))) \qquad \mathsf{Bij} \triangleq \textsc{Auth}(\text{finset}(\mathit{Loc} \times \mathit{Loc}))$$

Both are defined as authorative monoids which allow for having a global and a local part. To tie the two monoids together with a semantic region r (the name r is simply a positive integer) we use a third monoid:

$$\mathsf{Region} \triangleq \textsc{Auth}(\mathbb{Z}^+ \xrightarrow{\text{fin}} (\textsc{Ag}(\mathit{Names} \times \mathit{Names})))$$

We fix a global ghost name γ_{reg} for an instance of this last monoid. For Region, ownership of $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ indicates that the semantic region

r is represented by two ghost variables named γ_{bij} and γ_{rel}, for Bij and Rel respectively. Notice that this ownership of $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ is duplicable and also, due to the properties of the agreement monoid, we have that the semantic region tied to r is uniquely defined. Formally,

$$\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma'_{bij}, \gamma'_{rel})}^{\gamma_{reg}} \vdash \gamma_{bij} = \gamma'_{bij} \wedge \gamma_{rel} = \gamma'_{rel} \quad \text{(E.1)}$$

We can now present the $\mathsf{region}(r, \gamma_h, \gamma'_h)$ predicate in detail:

$$\mathsf{region}(r, \gamma_h, \gamma'_h) \triangleq \exists R, \gamma_{bij}, \gamma_{rel}.\ \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel}) : \mathsf{Region}}^{\gamma_{reg}} * \boxed{\bullet\, R : \mathsf{Rel}}^{\gamma_{rel}} * \ast_{(\ell,\ell') \mapsto \mathsf{ag}(\gamma_{pred}) \in R} \Big(\exists \Phi : (Val \times Val) \to iProp), v, v'.\ \ell \mapsto_{\gamma_h} v * \ell' \mapsto_{\gamma'_h} v' * \gamma_{pred} \Rrightarrow \Phi * \triangleright \Phi(v, v')\Big)$$

The predicate asserts that the semantic region r is associated with two ghost names, γ_{bij} and γ_{rel}, by $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, and full authoritative ownership of R, which is a mapping of pairs of locations to ghost names. Further, for each element $(\ell, \ell') \mapsto \mathsf{ag}(\gamma_{pred}) \in R$ we have ownership of the points-to predicates $\ell \mapsto_{\gamma_h} v$ and $\ell' \mapsto_{\gamma'_h} v'$ and the knowledge about a saved predicate Φ, named by γ_{pred}, that holds later for v and v'.

The regions predicate keeps track of all the allocated regions by having the full authoritative part $\boxed{\bullet\, M : \mathsf{Reg}}^{\gamma_{reg}}$:

$$\mathsf{regions} \triangleq \boxed{\exists M.\ \boxed{\bullet\, M : \mathsf{Region}}^{\gamma_{reg}} * \ast_{r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel}) \in M} \begin{pmatrix} \exists g : \mathrm{finset}(Loc \times Loc), \\ R : (Loc \times Loc) \xrightarrow{\mathrm{fin}} (\textsc{Ag}(Names)). \\ \boxed{\bullet\, g}^{\gamma_{bij}} * \mathsf{bijection}(g) * \boxed{\circ\, R}^{\gamma_{rel}} * g = \mathrm{dom}(R) \end{pmatrix}}$$

For each element $r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})$ in M, regions have full authoritative ownership of a bijection g and fragment ownership of R, which maps each pairs of locations to a ghost name for saved predicates. Here, g and the domain of R is forced to be equal, ensuring that all pairs that are related in the bijection are also related in the region. Notice that since the regions predicate is an invariant, it is also persistent.

Notice here as well that individual regions are tied to the regions predicate, regions, by having the fragment ownership of $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel}) : \mathsf{Region}}^{\gamma_{reg}}$ since the authoritative element $\boxed{\bullet\, M : \mathsf{Region}}^{\gamma_{reg}}$ is owned by regions. Similarly,

the regions predicate is tied to all regions by asserting ownership of the fragment $\boxed{\circ R}^{\gamma_{rel}}$. This illustrates how ghost resources are important to enforce relations in and out of invariants.

We can now give meaning to the abstract predicates used in the definition of `STRef` $\rho\ \tau$[1]:

$$\mathsf{isRgn}(\alpha, r) \triangleq \exists \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ \alpha = r * \boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$$

$$\mathsf{bij}(r, \ell, \ell') \triangleq \exists \gamma_{bij}, \gamma_{rel}.\ \boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ (\ell, \ell')}^{\gamma_{bij}}$$

$$\mathsf{rel}(r, \ell, \ell', \Phi) \triangleq \exists \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ \boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ [(\ell, \ell') \mapsto \mathsf{ag}(\gamma_{pred})]}^{\gamma_{bij}} * \gamma_{pred} \Rrightarrow \Phi$$

Each of the predicates owns the ghost resource suggested by its name. For instance, Property (6.5) from §6.3 can now be shown:

$$\mathsf{regions} \Rrightarrow \exists r.\ \mathsf{region}(r, \gamma_h, \gamma_h')$$

First, we open the invariant to obtain $\boxed{\bullet M : \mathsf{Region}}^{\gamma_{reg}}$. By Ghost-Alloc we obtain $\boxed{\bullet \emptyset \cdot \circ \emptyset : \mathsf{Rel}}^{\gamma_{rel}}$ and $\boxed{\bullet g}^{\gamma_{bij}}$, for fresh ghost names γ_{rel} and γ_{bij}. Now, by Auth-Alloc-Fpfn we can extend M with $r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})$, to obtain $\boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, for some r not in $\mathrm{dom}(M)$, since M is finite. $\mathsf{region}(r, \gamma_h, \gamma_h')$ now holds trivially, since there are no locations allocated in $\boxed{\bullet \emptyset}^{\gamma_{rel}}$. Similarly, $\mathsf{bijection}(\emptyset)$ and $\mathrm{dom}(\emptyset) = \emptyset$ hold trivially, so we have reestablished the body of the invariant.

E.2 Formalization in Coq

We have formalized our technical development and proofs in the Iris implementation in Coq Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017; Krebbers, Timany, and Birkedal, 2017. The Iris implementation in Coq Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017 includes a model of Iris and proof of soundness of the Iris logic itself. The Iris Proof Mode (IPM) Krebbers, Timany, and Birkedal, 2017 allows users to carry out proofs inside Iris in much the same way as in Coq itself by providing facilities for working with the substructural contexts and modalities of Iris. We have used

[1] The predicate $\boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ appears in all the abstract predicates to obtain γ_{bij} and γ_{rel}. This is to keep the initial description of the predicates simple. The redundancy does not exist in the actual implementation.

Iris and IPM to formalize the future modality, the IC predicates, our logical relation and to prove the state-independence theorem and all the refinements presented in this paper.

The Trusted Computing Base

Even though our logical relation has been defined inside the Iris logic, the soundness theorem of Iris (Krebbers, Jung, Bizjak, Jourdan, Dreyer, and Birkedal, 2017) allows us to prove the soundness of our logical relation:

```
Theorem binary_soundness Γ e e' τ : typed Γ e τ → typed Γ e' τ →
  (∀ Σ `{ICG_ST Σ} `{LogRelG Σ}, Γ ⊨ e ≤log≤ e' : τ)
  → Γ ⊨ e ≤ctx≤ e' : τ.
```

This statement says that whenever $\Xi \mid \Gamma \vdash e : \tau$ and $\Xi \mid \Gamma \vdash e' : \tau$ and we can prove in the Iris logic (notice the quantification of Iris parameters, `Σ `{ICG_ST Σ} `{LogRelG Σ}`)[2] that e and e' are logically related, then e contextually refines e'. Notice that Ξ does not appear in the Coq code as we are using de Bruijn indices to represent type variables and hence need no type level context. The definition of contextual refinement and well-typedness are in turn normal Coq statements, independent of Iris.

All lemmas and theorems in this paper are type checked by Coq without any assumptions or axioms apart from the use of functional extensionality which is used for the de Bruijn indices. It is used by the Autosubst library.

Extending Iris and IPM and instantiating them with STLang

The implementations of Iris and IPM in Coq are almost entirely independent of the choice of programming language. In practice, the only definitions that are parameterized by a language are the definitions of weakest-precondition and Hoare triples. To use these with a particular programming language, one needs to instantiate a data structure in Coq that represents the language. Basically, one is required to instantiate this data structure with the language's set of states (heaps in our case), expressions, values and reduction relation, together with proofs that they behave as expected (e.g., values do not reduce any further). In this work we use IC predicates and IC triples instead of the weakest precondition and Hoare triples used in earlier work. Therefore, we have also parameterized IC predicates and IC triples by a data structure representing the programming language. We instantiate these with STLang.

[2] Σ is the set of Iris resources and the other two parameters express that resources necessary for IC and our logical relations are present in Σ.

The formalization of Iris in Coq is a shallow embedding. That is, the model of the Iris logic is formalized in Coq, and terms of the type *iProp* (propositions of Iris) are defined as well-behaved predicates over the elements of that model. The advantage of shallow embeddings is that one can easily introduce new connectives and modalities to the logic by defining another function with *iProp* as co-domain. For instance, our IC predicate is defined as follows in Coq.

```
Definition ic_def {Λ Σ} `{ICState Λ, ICG Λ Σ} γ E e Φ : iProp Σ :=
  (∀ σ1 σ2 v n, (⌜nsteps pstep n (e, σ1) (of_val v, σ2)⌝
   ∗ ownP_full γ σ1) -∗ |≫{E}=[n]⇒  Φ v n ∗ ownP_full γ σ2)%I.
```

Here, $\ulcorner \cdot \urcorner$ embeds Coq propositions into Iris and `ownP_full` γ σ is the full ownership of the physical state of the language (parameter Λ), equivalent to our $heap_\gamma(\sigma)$. The `%I` at the end instructs Coq to parse connectives (e.g., the universal quantification) as Iris connectives and not those of Coq.

As discussed in (Krebbers, Timany, and Birkedal, 2017), IPM tactics, like the `iMod` tactic for elimination of modalities, simply apply lemmas with side conditions that are discharged with the help of Coq's type class inference mechanism. Extending IPM with support for the future modality and IC predicates essentially boils down to instantiating some of these type classes appropriately.

Representing binders

We use de Bruijn indices to represent variables both at the term level and the type level; in particular, we use the Autosubst library Schäfer, Tebbi, and Smolka, 2015. It provides excellent support for manipulating and simplifying terms with de Bruijn indices in Coq. The simplification procedure, however, seems to be non-linear in the size of the term. This is the main reason for the slowness of Coq's processing of our proofs.[3]

[3] About 17 minutes on a laptop using "make -j4" to compile our Coq formalization of about 12,500 lines.

Appendix F

Details of Relational Verification of Concurrent Programs with Continuation

F.1 Weakest precondition rules

REC-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e[\texttt{rec}\, f(x) = e, v/f, x]]\, \{\Phi\}}{\mathsf{wp}\, K[(\texttt{rec}\, f(x) = e)\ v]\, \{\Phi\}}$$

TAPP-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e]\, \{\Phi\}}{\mathsf{wp}\, K[(\Lambda\, e)\ \_]\, \{\Phi\}}$$

UNFOLD-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[v]\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{unfold}\, (\texttt{fold}\, v)]\, \{\Phi\}}$$

IF-TRUE-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e]\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{if true then}\, e\, \texttt{else}\, e']\, \{\Phi\}}$$

IF-FALSE-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e']\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{if false then}\, e\, \texttt{else}\, e']\, \{\Phi\}}$$

FST-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[v]\, \{\Phi\}}{\mathsf{wp}\, K[\pi_1\, (v, w)]\, \{\Phi\}}$$

SND-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[w]\, \{\Phi\}}{\mathsf{wp}\, K[\pi_2\, (v, w)]\, \{\Phi\}}$$

MATCH-INJ1-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e_1[v/x]]\, \{\Phi\}}{\mathsf{wp}\, K[\texttt{match inj}_1\, v\, \texttt{with inj}_1\, x \Rightarrow e_1 \mid \texttt{inj}_2\, x \Rightarrow e_2\, \texttt{end}]\, \{\Phi\}}$$

MATCH-INJ2-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e_2[v/x]]\,\{\Phi\}}{\mathsf{wp}\, K[\mathtt{match}\ \mathtt{inj}_2\, v\ \mathtt{with}\ \mathtt{inj}_1\, x \Rightarrow e_1 \mid \mathtt{inj}_2\, x \Rightarrow e_2\ \mathtt{end}]\,\{\Phi\}}$$

ALLOC-WP
$$\frac{\forall \ell.\ \ell \mapsto_i v \mathrel{-\!\!*} \mathsf{wp}\, K[\ell]\,\{\Phi\}}{\mathsf{wp}\, K[\mathtt{ref}(v)]\,\{\Phi\}}$$

LOAD-WP
$$\frac{\ell \mapsto_i v \mathrel{-\!\!*} \mathsf{wp}\, K[v]\,\{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[!\,\ell]\,\{\Phi\}}$$

STORE-WP
$$\frac{\ell \mapsto_i w \mathrel{-\!\!*} \triangleright\, \mathsf{wp}\, K[()]\,\{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[\ell \leftarrow w]\,\{\Phi\}}$$

CAS-SUC-WP
$$\frac{\ell \mapsto_i w \mathrel{-\!\!*} \triangleright\, \mathsf{wp}\, K[\mathtt{true}]\,\{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[\mathtt{CAS}(\ell, v, w)]\,\{\Phi\}}$$

CAS-FAIL-WP
$$\frac{\ell \mapsto_i v' \mathrel{-\!\!*} \triangleright\, \mathsf{wp}\, K[\mathtt{false}]\,\{\Phi\} \qquad \triangleright\, \ell \mapsto_i v' \qquad v \neq v'}{\mathsf{wp}\, K[\mathtt{CAS}(\ell, v, w)]\,\{\Phi\}}$$

FORK-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[()]\,\{\Phi\} \qquad \mathsf{wp}\, e\,\{\_.\ \top\}}{\mathsf{wp}\, K[\mathtt{fork}\ \{e\}]\,\{\Phi\}}$$

CALLCC-WP
$$\frac{\triangleright\, \mathsf{wp}\, K[e[\mathtt{cont}(K)/x]]\,\{\Phi\}}{\mathsf{wp}\, K[\mathtt{call/cc}\,(x.\, e)]\,\{\Phi\}}$$

THROW-WP
$$\frac{\triangleright\, \mathsf{wp}\, K'[v]\,\{\Phi\}}{\mathsf{wp}\, K[\mathtt{throw}\, v\, \mathtt{to}\, \mathtt{cont}(K')]\,\{\Phi\}}$$

F.2 Rules for execution on the specification side

REC-STEP
$$\frac{j \Mapsto K[(\mathtt{rec}\, f(x) = e)\ v]}{\Rrightarrow j \Mapsto K[e[\mathtt{rec}\, f(x) = e, v/f, x]]}$$

TAPP-STEP
$$\frac{j \Mapsto K[(\Lambda\, e)\ \_]}{\Rrightarrow j \Mapsto K[e]}$$

UNFOLD-STEP
$$\frac{j \Mapsto K[\mathtt{unfold}\,(\mathtt{fold}\, v)]}{\Rrightarrow j \Mapsto K[v]}$$

IF-TRUE-STEP
$$\frac{j \Mapsto K[\mathtt{if}\ \mathtt{true}\ \mathtt{then}\, e\ \mathtt{else}\, e']}{\Rrightarrow j \Mapsto K[e]}$$

IF-FALSE-STEP
$$\frac{j \Mapsto K[\mathtt{if}\ \mathtt{false}\ \mathtt{then}\, e\ \mathtt{else}\, e']}{\Rrightarrow j \Mapsto K[e']}$$

FST-STEP
$$\frac{j \Mapsto K[\pi_1\,(v, w)]}{\Rrightarrow j \Mapsto K[v]}$$

SND-STEP
$$\frac{j \Mapsto K[\pi_2\,(v, w)]}{\Rrightarrow j \Mapsto K[w]}$$

MATCH-INJ1-STEP
$$\frac{j \Rightarrow K[\texttt{match inj}_1\, v\, \texttt{with inj}_1\, x \Rightarrow e_1 \mid \texttt{inj}_2\, x \Rightarrow e_2\, \texttt{end}]}{\models\!\Rrightarrow j \Rightarrow K[e_1[v/x]]}$$

MATCH-INJ2-STEP
$$\frac{j \Rightarrow K[\texttt{match inj}_2\, v\, \texttt{with inj}_1\, x \Rightarrow e_1 \mid \texttt{inj}_2\, x \Rightarrow e_2\, \texttt{end}]}{\models\!\Rrightarrow j \Rightarrow K[e_2[v/x]]}$$

ALLOC-STEP
$$\frac{j \Rightarrow K[\texttt{ref}(v)]}{\models\!\Rrightarrow \exists \ell.\, \ell \mapsto_s v * j \Rightarrow K[\ell]}$$

LOAD-STEP
$$\frac{\ell \mapsto_s v \qquad j \Rightarrow K[!\,\ell]}{\models\!\Rrightarrow \exists \ell.\, \ell \mapsto_s v * j \Rightarrow K[v]}$$

STORE-STEP
$$\frac{\ell \mapsto_s v \qquad j \Rightarrow K[\ell \leftarrow w]}{\models\!\Rrightarrow \ell \mapsto_s w * j \Rightarrow K[()]}$$

CAS-SUC-STEP
$$\frac{\ell \mapsto_s v \qquad j \Rightarrow K[\texttt{CAS}(\ell, v, w)]}{\models\!\Rrightarrow \ell \mapsto_s w * j \Rightarrow K[\texttt{true}]}$$

CAS-FAIL-STEP
$$\frac{\ell \mapsto_s v' \qquad j \Rightarrow K[\texttt{CAS}(\ell, v, w)] \qquad v \neq v'}{\models\!\Rrightarrow \ell \mapsto_s v' * j \Rightarrow K[\texttt{false}]}$$

FORK-STEP
$$\frac{j \Rightarrow K[\texttt{fork}\ \{e\}]}{\models\!\Rrightarrow j \Rightarrow K[()] * \exists j'.\, j' \Rightarrow e}$$

CALLCC-STEP
$$\frac{j \Rightarrow K[\texttt{call/cc}\,(x.\, e)]}{\models\!\Rrightarrow j \Rightarrow K[e[\texttt{cont}(K)/x]]}$$

THROW-STEP
$$\frac{j \Rightarrow K[\texttt{throw}\, v\, \texttt{to}\, K']}{\models\!\Rrightarrow j \Rightarrow K'[v]}$$

F.3 Context-local Weakest precondition rules

REC-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\, e[\texttt{rec}\, f(x) = e, v/f, x]\, \{\Phi\}}{\mathsf{clwp}\, (\texttt{rec}\, f(x) = e)\, v\, \{\Phi\}}$$

TAPP-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\, e\, \{\Phi\}}{\mathsf{clwp}\, (\Lambda\, e)\, \_\, \{\Phi\}}$$

UNFOLD-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\, v\, \{\Phi\}}{\mathsf{clwp}\, \texttt{unfold}\, (\texttt{fold}\, v)\, \{\Phi\}}$$

IF-TRUE-CLWP
$$\frac{o}{g} \qquad relcc - app - if - true -$$

$$clwp \triangleright \mathsf{clwp}\, e\, \{\Phi\} \mathsf{clwp}\, K[\texttt{if true then}\, e\, \texttt{else}\, e']\, \{\Phi\}$$

IF-FALSE-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\, e'\, \{\Phi\}}{\mathsf{clwp}\, \texttt{if false then}\, e\, \texttt{else}\, e'\, \{\Phi\}}$$

FST-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\, v\, \{\Phi\}}{\mathsf{clwp}\, \pi_1\, (v, w)\, \{\Phi\}}$$

SND-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\, w\, \{\Phi\}}{\mathsf{clwp}\, \pi_2\, (v, w)\, \{\Phi\}}$$

MATCH-INJ1-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e_1[v/x]\ \{\Phi\}}{\mathsf{clwp}\ \mathtt{match}\ \mathtt{inj}_1\ v\ \mathtt{with}\ \mathtt{inj}_1\ x \Rightarrow e_1 \mid \mathtt{inj}_2\ x \Rightarrow e_2\ \mathtt{end}\ \{\Phi\}}$$

MATCH-INJ2-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e_2[v/x]\ \{\Phi\}}{\mathsf{clwp}\ \mathtt{match}\ \mathtt{inj}_2\ v\ \mathtt{with}\ \mathtt{inj}_1\ x \Rightarrow e_1 \mid \mathtt{inj}_2\ x \Rightarrow e_2\ \mathtt{end}\ \{\Phi\}}$$

ALLOC-CLWP
$$\mathsf{clwp}\ K[\mathtt{ref}(v)]\ \{v.\ \exists \ell.\ v = \ell * \ell \mapsto_i v\}$$

LOAD-CLWP
$$\frac{\triangleright\, \ell \mapsto_i v}{\mathsf{clwp}\ !\ell\ \{w.\ w = v * \ell \mapsto_i v\}}$$

STORE-CLWP
$$\frac{\triangleright\, \ell \mapsto_i v}{\mathsf{clwp}\ \ell \leftarrow w\ \{v'.\ v' = () * \ell \mapsto_i w\}}$$

CAS-SUC-WP
$$\frac{\triangleright\, \ell \mapsto_i v}{\mathsf{clwp}\ \mathtt{CAS}(\ell, v, w)\ \{v'.\ v\ \ = \mathtt{true} * \ell \mapsto_i w\}}$$

CAS-FAIL-CLWP
$$\frac{\triangleright\, \ell \mapsto_i v' \qquad v \neq v'}{\mathsf{clwp}\ \mathtt{CAS}(\ell, v, w)\ \{v''.\ v'' = \mathtt{false} * \ell \mapsto_i v'\}}$$

FORK-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e\ \{\_.\ \top\}}{\mathsf{clwp}\ \mathtt{fork}\ \{e\}\ \{v.\ v = ()\}}$$

F.4 Logical Relations

Observational refinement ($\mathcal{O} : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$):

$$\mathcal{O}(e, e') \triangleq \forall j.\ j \Rightarrow e' \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\exists w.\ j \Rightarrow w\}$$

Value interpretation of types ($[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ for $\Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$):

$$[\![\Xi \vdash X]\!]_\Delta \triangleq \Delta(X)$$

$$[\![\Xi \vdash 1]\!]_\Delta(v, v') \triangleq v = v' = ()$$

$$[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v, v') \triangleq v = v' = \mathtt{true} \lor v = v' = \mathtt{false}$$

$$[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v, v') \triangleq \exists n.\ v = v' = n$$

$$[\![\Xi \vdash \tau_1 + \tau_2]\!]_\Delta(v, v') \triangleq \bigvee_{i\in\{1,2\}} \left(\begin{array}{c} \exists w, w'.\ v = \mathtt{inj}_i\, w \wedge v' = \mathtt{inj}_i\, w' \wedge \\ [\![\Xi \vdash \tau_i]\!]_\Delta(w, w') \end{array} \right)$$

$$[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v) \triangleq \exists w_1, w_2, w_1', w2'.\ v = (w_1, w_2) \wedge v' = (w_1', w_2') \wedge [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') * [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')$$

$$[\![\Xi \vdash \mu X.\ \tau]\!]_\Delta(v, v') \triangleq \mu f : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}.\ \exists w, w'.\ v = \mathtt{fold}\, w \wedge v' = \mathtt{fold}\, w' \wedge \triangleright [\![X, \Xi \vdash \tau]\!]_{\Delta, X\mapsto f}(w, w')$$

$$[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') \triangleq \square\left(\forall w, w'.\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow [\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(v\ w, v'\ w')\right)$$

$$[\![\Xi \vdash \forall X.\ \tau]\!]_\Delta(v, v') \triangleq \square\left(\forall f : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}.\ \mathsf{persistent}(f) \Rightarrow [\![X, \Xi \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X\mapsto f}(v\ \_, v'\ \_)\right)$$

$$[\![\Xi \vdash \mathtt{ref}(\tau)]\!]_\Delta(v, v') \triangleq \exists \ell, \ell'.\ v = \ell \wedge v' = \ell' \wedge \boxed{\exists w, w'.\ \ell \mapsto_i w * \ell' \mapsto_s w' * [\![\Xi \vdash \tau]\!]_\Delta(w, w')}^{\mathcal{N}.\ell.\ell'}$$

$$[\![\Xi \vdash \mathtt{cont}(\tau)]\!]_\Delta(v, v) \triangleq \exists K, K'.\ v = \mathtt{cont}(K) \wedge v' = \mathtt{cont}(K') \wedge \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K')$$

Evaluation context interpretation of types ($\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Ectx} \times \mathit{Ectx} \to \mathit{iProp}$ for $\Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$):

$$\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K') \triangleq \forall v, v'.\ [\![\Xi \vdash \tau]\!]_\Delta(v, v') \Rightarrow \mathcal{O}(K[v], K'[v'])$$

Expression interpretation of types ($[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$ for $\Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$):

$$[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e, e') \triangleq \forall K, K'.\ \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K') \Rightarrow \mathcal{O}(K[e], K'[e'])$$

Logical relatedness ($\Xi \mid \Gamma \vDash e \le_{\log} e' : \tau : \mathit{iProp}$):

$$\Xi \mid \Gamma \vDash e \le_{\log} e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v'}.\ \left(\ast_{x_i:\tau_i} [\![\Xi \vdash \tau_i]\!]_\Delta(v_i, v_i')\right) \Rightarrow [\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$

assuming $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$

F.5 Inadmissibility of the Bind Rule

Consider the derivation given the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{false} = \texttt{false}}}{\mathsf{wp}\,\texttt{false}\,\{v.\ v = \texttt{false}\}}}{\mathsf{wp}\,\texttt{if}\,\texttt{true}\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true}\,\{v.\ v = \texttt{false}\}}}{\begin{array}{l}\mathsf{wp}\,\texttt{true}\\ \{w.\ \mathsf{wp}\,\texttt{if}\,w\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true}\,\{v.\ v = \texttt{false}\}\}\end{array}}}{\begin{array}{l}\mathsf{wp}\,\texttt{throw}\,\texttt{true}\,\texttt{to}\,-\\ \{w.\ \mathsf{wp}\,\texttt{if}\,w\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true}\,\{v.\ v = \texttt{false}\}\}\end{array}}\,\text{INADMISSIBLE-BIND}}{\begin{array}{l}\mathsf{wp}\,\texttt{if}\,(\texttt{throw}\,\texttt{true}\,\texttt{to}\,-)\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true}\\ \{v.\ v = \texttt{false}\}\end{array}}}{\begin{array}{l}\mathsf{wp}\,\texttt{call/cc}\,(x.\,\texttt{if}\,(\texttt{throw}\,\texttt{true}\,\texttt{to}\,x)\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true})\\ \{v.\ v = \texttt{false}\}\end{array}}$$

Here, we omit steps corresponding to eliminations of $\triangleright$. Note that we can easily show that

$$\texttt{call/cc}\,(x.\,\texttt{if}\,(\texttt{throw}\,\texttt{true}\,\texttt{to}\,x)\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true})$$

reduces to the value `true` which falsifies the derivation above.

F.5.1 The resource for one-shot bit

For details of representing resources with monoids in Iris see Appendix C. The predicates for representing the one-shot bits, $\mathit{OneShotBits}(M)$ and $\mathit{isOneShotBit}(b)$, are defined as follows:

$$\textsc{OneShotBit} \triangleq \textsc{Auth}(\mathrm{finset}(\ell))$$

$$\mathit{OneShotBits}(M) \triangleq \boxed{\bullet M}^{\gamma_{os}}$$

$$\mathit{isOneShotBit}(b) \triangleq \boxed{\circ \{b\}}^{\gamma_{os}}$$

It is easy to see, based on the rules for monoids in Appendix C, that $\mathit{isOneShotBit}(b)$ is persistent and that:

$$\mathit{OneShotBits}(M) * \mathit{isOneShotBit}(b) \vdash \checkmark(\bullet M \cdot \circ \{b\}) \vdash \{b\} \subseteq M \vdash b \in M$$

Bibliography

Aczel, Peter (1977). "An Introduction to Inductive Definitions". In: *Studies in Logic and the Foundations of Mathematics* 90. Handbook of Mathematical Logic, pp. 739 –782.

Aczel, Peter (1999). "On Relating Type Theories and Set Theories". In: *Types for Proofs and Programs: International Workshop, TYPES' 98, Selected Papers*, pp. 1–18.

Ahmed, Amal J., Andrew W. Appel, and Roberto Virga (2002). "A Stratified Semantics of General References Embeddable in Higher-Order Logic". In: *LICS*, pp. 75–86.

Ahmed, Amal (2004). "Semantics of Types for Mutable State". PhD thesis. Princeton University.

Ahmed, Amal (2006). "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". In: *ESOP*.

Ahmed, Amal, Derek Dreyer, and Andreas Rossberg (2009). "State-Dependent Representation Independence". In: *POPL*.

Ahrens, Benedikt, Krzysztof Kapulkin, and Michael Shulman (2015). "Univalent categories and the Rezk completion". In: *Mathematical Structures in Computer Science* 25.05, pp. 1010–1039. URL: https://github.com/benediktahrens/rezk_completion.

Appel, Andrew and David McAllester (2001). "An Indexed Model of Recursive Types for Foundational Proof-Carrying Code". In: *TOPLAS* 23.5, pp. 657–683.

Appel, Andrew, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon (2007). "A Very Modal Model of a Modern, Major, General Type System". In: *POPL*.

Awodey, Steve (2010). *Category theory*. Oxford University Press.

Barras, Bruno (2012). *Semantical Investigation in Intuitionistic Set Theory and Type Theoris with Inductive Families*. Habilitation thesis, University Paris Diderot – Paris 7.

Benton, Nick and Peter Buchlovsky (2007). "Semantics of an effect analysis for exceptions". In: *TLDI*.

Benton, Nick and Chung-Kil Hur (2009). "Biorthogonality, Step-Indexing and Compiler Correctness". In: *ICFP*.

Benton, Nick, Andrew Kennedy, Lennart Beringer, and Martin Hofmann (2007). "Relational semantics for effect-based program transformations with dynamic allocation". In: *PPDP*.

Benton, Nick, Andrew Kennedy, Lennart Beringer, and Martin Hofmann (2009). "Relational semantics for effect-based program transformations: higher-order store". In: *PPDP*.

Benton, Nick, Andrew Kennedy, Martin Hofmann, and Lennart Beringer (2006). "Reading, writing and relations". In: *PLAS*. Springer.

Berger, Martin (2010). "Program Logics for Sequential Higher-Order Control". In: *FSEN 2009, Revised Selected Papers*, pp. 194–211.

Bertot, Yves and Pierre Castéran (2013). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions (Texts in Theoretical Computer Science. An EATCS Series)*. Springer. ISBN: 978-3-662-07964-5. URL: https://www.springer.com/gp/book/9783540208549.

Biering, Bodil, Lars Birkedal, and Noah Torp-Smith (Aug. 2007). "BI-hyperdoctrines, Higher-order Separation Logic, and Abstraction". In: *ACM Trans. Program. Lang. Syst.* 29.5.

Birkedal, Lars and Aleš Bizjak (2017). "Iris Lecture Notes". Manuscript. URL: http://iris-project.org/tutorial-material.html.

Birkedal, Lars, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring (2011). "First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees". In: *LICS*.

Birkedal, Lars, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang (2011). "Step-Indexed Kripke Models over Recursive Worlds". In: *POPL*.

Birkedal, Lars, Filip Sieczkowski, and Jacob Thamsborg (2012). "A Concurrent Logical Relation". In: *CSL*.

Birkedal, Lars, Kristian Støvring, and Jacob Thamsborg (2009). "Realizability Semantics of Parametric Polymorphism, General References, and Recursive Types". In: *FOSSACS*.

Birkedal, Lars, Kristian Støvring, and Jacob Thamsborg (2010). "The category-theoretic solution of recursive metric-space equations". In: *Theoretical Computer Science* 411.47, pp. 4102–4122.

Boulier, Simon, Pierre-Marie Pédrot, and Nicolas Tabareau (Jan. 2017). "The Next 700 Syntactical Models of Type Theory". In: *CPP*. Paris, France, pp. 182–194.

Cerone, Andrea, Alexey Gotsman, and Hongseok Yang (2014). "Parameterised Linearisability". In: *ICALP*.

Chlipala, Adam (2013). *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant (MIT Press)*. The MIT Press. ISBN: 9780262026659. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

Conference ranking portal of CORE. URL: http://portal.core.edu.au/conf-ranks/.

Coquand, T. (May 1986). *An analysis of Girard's paradox*. Tech. rep. RR-0531. INRIA. URL: https://hal.inria.fr/inria-00076023.

Coquand, Thierry and Gérard Huet (1988). "The calculus of constructions". In: *Information and computation* 76.2-3, pp. 95–120.

Crolard, T. and E. Polonowski (2012). "Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control". In: *The Journal of Logic and Algebraic Programming* 81.3. The 22nd Nordic Workshop on Programming Theory (NWPT 2010), pp. 181 –208. ISSN: 1567-8326.

Curry, Haskell B (1934). "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences* 20.11, pp. 584–590.

Damas, Luís (1984). "Type assignment in programming languages". PhD thesis. University of Edinburgh, UK.

Danvy, Olivier and Andrzej Filinski (1990). "Abstracting Control". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*.

Delbianco, Germán Andrés and Aleksandar Nanevski (2013). "Hoare-style reasoning with (algebraic) continuations". In: *ACM SIGPLAN Notices* 48.9, pp. 363–376.

Devriese, Dominique, Marco Patrignani, and Frank Piessens (2016). "Fully-abstract compilation by approximate back-translation". In: *ACM SIGPLAN Notices*, pp. 164–177.

Dinsdale-Young, T., M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis (2010). "Concurrent abstract predicates". In: *ECOOP*, pp. 504–528.

Dinsdale-Young, Thomas, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang (2013). "Views: Compositional reasoning for concurrent programs". In: *POPL*.

Drake, Frank R (1974). *Set theory : an introduction to large cardinals*. Studies in logic and the foundations of mathematics 76. Amsterdam: North-Holland. ISBN: 0720422795.

Dreyer, D., A. Ahmed, and L. Birkedal (2011). "Logical Step-Indexed Logical Relations". In: *LMCS* 7.2:16.

Dreyer, Derek, Amal Ahmed, and Lars Birkedal (2009). "Logical Step-Indexed Logical Relations". In: *LICS*.

Dreyer, Derek, Georg Neis, and Lars Birkedal (2010). "The Impact of Higher-Order State and Control Effects on Local Relational Reasoning". In: *ICFP*.

Dreyer, Derek, Georg Neis, and Lars Birkedal (2012). "The Impact of Higher-Order State and Control Effects on Local Relational Reasoning". In: *JFP* 22.4–5, pp. 477–528.

Dybjer, Peter (1991). "Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics". In: *Logical Frameworks*. Cambridge: Cambridge University Press, pp. 280–306.

Felleisen, Matthias and Robert Hieb (1992). "The revised report on the syntactic theories of sequential control and state". In: *TCS* 103.2, pp. 235–271.

Felleisen, Mattias (1988). "The Theory and Practice of First-class Prompts". In: *POPL*.

Flatt, Matthew (2017). *More: Systems Programming with Racket*. https://docs.racket-lang.org/more/index.html.

Friedman, Daniel and Christopher Haynes (1985). "Constraining control". In: *POPL*.

Gifford, D. K. and J. M. Lucassen (1986). "Integrating functional and imperative programming". In: *LISP*.

Girard, Jean-Yves (1972). "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris VII.

Gitik, M. (1980). "All uncountable cardinals can be singular". In: *Israel Journal of Mathematics* 35.1, pp. 61–88.

Gross, Jason, Adam Chlipala, and David I. Spivak (2014a). "Experience Implementing a Performant Category-Theory Library in Coq". In: *ITP*, pp. 275–291.

Gross, Jason, Adam Chlipala, and David I. Spivak (2014b). *Experience Implementing a Performant Category-Theory Library in Coq*. eprint: arXiv:1401.7694.

Harper, Robert (1994). "A simplified account of polymorphic references". In: *Information Processing Letters* 51.4, pp. 201 –206.

Harper, Robert and Robert Pollack (1991). "Type Checking with Universes". In: *TCS* 89.1, pp. 107–136.

Hendershott, Greg. *Written in Racket*. URL: http://www.greghendershott.com/2014/09/written-in-racket.html.

Herlihy, Maurice P. and Jeannette M. Wing (1990). "Linearizability: a correctness condition for concurrent objects". In: *TOPLAS* 12.3, pp. 463–492.

Howard, William A (1980). "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44. original paper manuscript from 1969, pp. 479–490.

Huet, Gérard P. and Amokrane Saïbi (2000). "Constructive category theory". In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pp. 239–276. URL: http://www.lix.polytechnique.fr/coq/pylons/coq/pylons/contribs/view/ConCaT/v8.4.

Huet, Gérard, Gilles Kahn, and Christine Paulin-Mohring (2018). *The Coq Proof Assistant A Tutorial*. URL: https://coq.inria.fr/distrib/current/files/Tutorial.pdf.

Hurkens, Antonius JC (1995). "A simplification of Girard's paradox". In: *International Conference on Typed Lambda Calculi and Applications*. Edinburgh, UK: Springer, pp. 266–278.

Jacobs, Bart (1999). *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland.

Jacobs, Bart (2013). *The Essence of Coq as a Formal System*. URL: https://people.cs.kuleuven.be/~bart.jacobs/coq-essence.pdf.

Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer (Dec. 2017). "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL, 66:1–66:34.

Jung, Ralf, Robbert Krebbers, Lars Birkedal, and Derek Dreyer (2016). "Higher-order ghost state". In: *ICFP*, pp. 256–269.

Jung, Ralf, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer (2018). "Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic". In: *the Journal of Functional Programming (JFP)*, (Accepted for publication).

Jung, Ralf, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer (2015). "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *POPL*, pp. 637–650.

Krebbers, Robbert, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal (2017). "The Essence of Higher-Order Concurrent Separation Logic". In: *ESOP*.

Krebbers, Robbert, Amin Timany, and Lars Birkedal (2017). "Interactive Proofs in Higher-Order Concurrent Separation Logic". In: *POPL*.

Krishnamurthi, Shriram, Peter Walton Hopkins, Jay McCarthy, Paul T Graunke, Greg Pettyjohn, and Matthias Felleisen (2007). "Implementation and use of the PLT Scheme web server". In: *Higher-Order and Symbolic Computation* 20.4, pp. 431–460.

Krogh-Jespersen, Morten, Kasper Svendsen, and Lars Birkedal (2017). "A Logical Account of a Type-and-Effect System". In: *POPL*.

Laird, James (1997). "Full Abstraction for Functional Languages with Control". In: *LICS*.

Lambek, Joachim and Philip J Scott (1988). *Introduction to higher-order categorical logic*. Vol. 7. Cambridge University Press.

Landin, P. J. (1964). "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4, pp. 308–320.

Launchbury, John and Simon L. Peyton Jones (1994). "Lazy Functional State Threads". In: *PLDI '94*, pp. 24–35.

Launchbury, John and Simon L. Peyton Jones (1995). "State in haskell". In: *Lisp and symbolic computation* 8.4, pp. 293–341.

Lee, Gyesik and Benjamin Werner (2011). "Proof-irrelevant model of CC with predicative induction and judgmental equality". In: *LMCS* 7.4.

Ley-Wild, Ruy and Aleksandar Nanevski (2013). "Subjective Auxiliary State for Coarse-Grained Concurrency". In: *POPL*.

Mac Lane, Saunders (1978). *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media.

McBride, Conor (2015). *Universe hierarchies*. Blog post. URL: https://pigworker.wordpress.com/2015/01/09/universe-hierarchies/.

McLarty, Colin (1996). *Elementary Categories, Elementary Toposes*. Oxford, UK: Oxford University Press. ISBN: 0-19-851473-5.

Megacz, Adam (2011). *Category Theory in Coq*. URL: http://www.megacz.com/berkeley/coq-categories/.

Might, Matt (2017). *Low-level web programming in Racket + a wiki in 500 lines*. URL: http://matt.might.net/articles/low-level-web-in-racket/.

Milner, Robin (1978). "A Theory of Type Polymorphism in Programming Languages". In: *Journal of Computer and System Science* 17.3, pp. 348–375.

Miquel, Alexandre and Benjamin Werner (2003). "The Not So Simple Proof-Irrelevant Model of CC". In: *TYPES 2002, Selected Papers*, pp. 240–258.

Mitchell, John C. (1996). *Foundations of Programming Languages*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-13321-0.

Moggi, E. and Amr Sabry (Nov. 2001). "Monadic Encapsulation of Effects: A Revised Approach (Extended Version)". In: *J. Funct. Program.* 11.6, pp. 591–627.

Murawski, Andrzej S. and Nikos Tzevelekos (2017). "Higher-Order Linearisability". In: *CONCUR 2017*.

Nahas, Mike (2012). *A tutorial by Mike Nahas*. URL: https://coq.inria.fr/tutorial-nahas.

Nanevski, Aleksandar, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco (2014). "Communicating State Transition Systems for Fine-Grained Concurrent Resources". In: *ESOP*, pp. 290–310.

O'Hearn, Peter W. (2007). "Resources, Concurrency and Local Reasoning". In: *Theor. Comput. Sci.* 375.1-3, pp. 271–307.

Paulin-Mohring, C. (Dec. 1996). "Définitions Inductives en Théorie des Types d'Ordre Supérieur". Habilitation à diriger les recherches. Université Claude Bernard Lyon I.

Peebles, Daniel, James Deikun, Ulf Norell, Dan Doel, Andrea Vezzosi, Darius Jahandarie, and James Cook (2016). *copumpkin/categories*. URL: https://github.com/copumpkin/categories.

Pierce, Benjamin C., Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent

Yorgey (2017). *Software Foundations*. Version 5.0. http://www.cis.upenn.edu/~bcpierce/sf. Electronic textbook.
Pitts, Andrew M. (1996). "Reasoning about Local Variables with Operationally-Based Logical Relations". In: *LICS*.
Pitts, Andrew (2005). "Typed Operational Reasoning". In: *Advanced Topics in Types and Programming Languages*. Ed. by B. C. Pierce. MIT Press. Chap. 7.
Plotkin, G and Martín Abadi (1993). "A logic for parametric polymorphism". In: *TLCA*.
Plotkin, Gordon D. (1977). "LCF considered as a programming language". In: *Theoretical computer science* 5.3, pp. 223–255.
Queinnec, Christian (2004). "Continuations and web servers". In: *Higher-Order and Symbolic Computation* 17.4, pp. 277–295.
Reynolds, John C. (1983). "Types, Abstraction, and Parametric Polymorphism". In: *Information Processing*.
Rouhling, Damien (2014). *Dependently typed lambda calculus with a lifting operator*. Tech. rep. Internship report. ENS Lyon.
Salibra, Antonino (2012). "Scott is Always Simple". In: *MFCS'12*. Berlin, Heidelberg: Springer-Verlag, pp. 31–45.
Schäfer, Steven, Tobias Tebbi, and Gert Smolka (2015). "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions". In: *ITP*, pp. 359–374.
Semmelroth, Miley and Amr Sabry (1999). "Monadic Encapsulation in ML". In: *ICFP '99*, pp. 8–17.
Sergey, Ilya, Aleksandar Nanevski, and Anindya Banerjee (2015). "Mechanized verification of fine-grained concurrent programs". In: *PLDI*, pp. 77–87.
Siles, Vincent and Hugo Herbelin (2012). "Pure Type System conversion is always typable". In: *J. Funct. Program.* 22.2, pp. 153–180.
Sozeau, Matthieu and Nicolas Tabareau (2014). "Universe Polymorphism in Coq". In: *ITP*, pp. 499–514.
Støvring, Kristian and Soren Lassen (2007). "A Complete, Co-Inductive Syntactic Theory of Sequential Control and State". In: *POPL*.
Svendsen, Kasper and Lars Birkedal (2014). "Impredicative Concurrent Abstract Predicates". In: *ESOP*, pp. 149–168.
Thamsborg, Jacob and Lars Birkedal (2011). "A Kripke Logical Relation for Effect-Based Program Transformations". In: *ICFP*.
The Coq Development Team (Dec. 2017). *The Coq Proof Assistant, version 8.7.1*. DOI: 10.5281/zenodo.1133970. URL: https://doi.org/10.5281/zenodo.1133970.
The Coq development team (2018). *The Coq Proof Assistant Reference Manual*. URL: https://coq.inria.fr/refman/.
The Univalent Foundations Program. *HoTT Version of Coq and Library*. URL: https://github.com/HoTT/HoTT.

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: http://homotopytypetheory.org/book.

Timany, Amin (2016b). *Categories-HoTT*. URL: https://github.com/amintimany/Categories-HoTT.

Timany, Amin (2016a). *Categories*. DOI: 10.5281/zenodo.50689. URL: https://github.com/amintimany/Categories/tree/FSCD16.

Timany, Amin and Bart Jacobs (2015). "First Steps Towards Cumulative Inductive Types in CIC". In: *ICTAC*, pp. 608–617.

Timany, Amin and Bart Jacobs (2016b). *Category Theory in Coq 8.5: Extended Version*. Tech. rep. CW697. iMinds-Distrinet, KU Leuven. URL: http://www2.cs.kuleuven.be/publicaties/rapporten/cw/CW697.abs.html.

Timany, Amin and Bart Jacobs (2016a). "Category Theory in Coq 8.5". In: *FSCD*. Porto, Portugal: LIPIcs, 30:1–30:18.

Timany, Amin and Bart Jacobs (2016c). *The Category-theoretic Solution of Recursive Ultra-metric Space Equations*. Presented at CoqPL'16: The Second International Workshop on Coq for PL. URL: https://github.com/amintimany/CTDT.

Timany, Amin and Matthieu Sozeau (2018). "Cumulative Inductive Types in Coq". In: *FSCD*. Accepted for publication. Oxford, UK: LIPIcs.

Timany, Amin, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal (Jan. 2018). "A Logical Relation for Monadic Encapsulation of State: Proving contextual equivalences in the presence of runST". In: *Proc. ACM Program. Lang.* 2.POPL.

Turon, Aaron, Derek Dreyer, and Lars Birkedal (2013). "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency". In: *ICFP*, pp. 377–390.

Turon, Aaron, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer (2013). "Logical relations for fine-grained concurrency". In: *POPL*.

Werner, Benjamin (1997). "Sets in types, types in sets". In: *TACS'97*, pp. 530–546.

Wright, A.K. and M. Felleisen (1994). "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1, pp. 38 –94.

Wright, Andrew K. (1995). "Simple imperative polymorphism". In: *LISP and Symbolic Computation* 8.4, pp. 343–355.

da Rocha Pinto, Pedro, Thomas Dinsdale-Young, and Philippa Gardner (2014). "TaDA: A Logic for Time and Data Abstraction". In: *ECOOP*, pp. 207–231.

de Bruijn, N. G. (1970). "The mathematical language AUTOMATH, its usage, and some of its extensions". In: *Symposium on Automatic Demonstration*, pp. 29–61.

The Coq development team (2015). *Coq 8.5 Reference Manual*. Inria.

Index

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
amin.timany@cs.kuleuven.be
https://distrinet.cs.kuleuven.be