IT UNIVERSITY OF COPENHAGEN

DOCTORAL THESIS

Denotational semantics in Synthetic Guarded Domain Theory

Author:
Marco PAVIOTTI

Supervisor:
Rasmus E. MØGELBERG
Co-supervisor:
Jesper BENGTSON

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Programming Languages and Semantics Group
Theoretical Computer Science Section

Defended on 13th October, 2016

Declaration of Authorship

I, Marco PAVIOTTI, declare that this thesis titled, "Denotational semantics in Synthetic Guarded Domain Theory" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the IT University of Copenhagen.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at the IT University of Copenhagen or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 13th October, 2016

IT UNIVERSITY OF COPENHAGEN

Abstract

Theoretical Computer Science Section

Doctor of Philosophy

Denotational semantics in Synthetic Guarded Domain Theory

by Marco PAVIOTTI

In functional programming, features such as recursion, recursive types and general references are central. To define semantics of this kind of languages one needs to come up with certain definitions which may be nontrivial to show well-defined. This is because they are circular. Domain theory has been used to solve this kind of problems for specific languages, unfortunately, this technique does not scale for more featureful languages, which prevented it from being widely used.

Step-indexing is a more general technique that has been used to break circularity of definitions. The idea is to tweak the definition by adding a well-founded structure that gives a handle for recursion. Guarded dependent Type Theory (gDTT) is a type theory which implements step-indexing via a unary modality used to guard recursive definitions. Every circular definition is well-defined as long as the recursive variable is *guarded*.

In this thesis we show that gDTT is a natural setting to give denotational semantics of typed functional programming languages with recursion and recursive types. We formulate operational semantics and denotational semantics and prove computational adequacy entirely *inside the type theory*. Furthermore, our interpretation is *synthetic*: types are interpreted as types in the type theory and programs as type-theoretical terms. Moreover, working directly in gDTT has advantages compared with existing set-theoretic models.

Finally, this work builds the foundations for doing denotational semantics of languages with much more challenging features, for example, of general references for which denotational techniques were previously beyond reach.

IT UNIVERSITET I KØBENHAVN

Resumé

Sektion for teretisk datalogi

Ph.D.-afhandling

Denotationel semantik i synthetic guarded domain theory

af Marco PAVIOTTI

Rekursivt definerede funktioner og rekursive typer er centrale begreber i funktionel programmering, og visse funktionelle sprog som f.eks. ML tillader også brug af generelle referencer. For at give semantik for disse konstruktioner er man nødt til at bruge rekursivt definerede objekter hvis eksistens det kan være svært at bevise. Domæneteori er en teori om ordnede mængder, i hvilken man kan give mening til en stor klasse af rekursive definitioner og bruge disse til at modellere programmeringssprog. På det seneste har domæneteori dog vist sig for kompliceret og utilstrækkelig til at blive anvendt til modeller af moderne programmeringssprog.

Step-indexing er en ny teknik til at bryde cirkulariteten i rekursive definitioner. Ideen er at tilføje elementer fra en velfunderet struktur til definitionen og at bruge disse til at tælle udfoldninger af den rekursive definition. Guarded Dependent Type Theory (gDTT) er en abstract teori for step-indexing i hvilken udfoldninger af rekursive definitioner bliver talt ved hjælp af en modal typekonstruktør. En rekursiv definition er veldefineret hvis blot alle forekomster af rekursionsvariablen er under modaliteten.

I denne afhandling viser vi at gDTT er et naturligt sprog for denotational semantik af typede funktionelle programmeringssprog med rekursive funktioner og typer. Vi konstruerer både den operationelle og denotationelle semantik i gDTT og beviser en stærk sammenhæng mellem disse (computational adequacy) i gDTT. Den denotationelle model er syntetisk i den forstand at typer og termer i objektsproget fortolkes som typer og termer i gDTT. Denne tilgang giver en simplere og mere elegant repræsentation af teorien end tidligere modeller defineret i mængdelære.

Denne afhandling lægger fundamentet for fremtidig forskning i denotationel semantik for programmeringssprog med avancerede konstruktioner som f.eks. generelle referencer, der ikke tidligere har været modelleret denotationelt.

Acknowledgements

First and foremost I would like to thank my supervisor Rasmus Møgelberg for being always available when I needed help of any kind. He has taught me a great deal during these years.

A big thank also to Lars Birkedal for hosting me seven months at Aarhus University. I had a wonderful and very productive time in the Logic and Semantics group thanks also to *all* its members. This experience changed me significantly.

I would like to thank all the people who directly contributed with useful comments on this thesis. My co-supervisor Jesper Bengtson who co-authored a paper in this thesis. Aleš Bizjak, Peter Brottveit Bock , Marco Carbone, Alec Faithfull, Jonas Frey, Tim Revell and João Costa Seco for useful comments on this thesis. As last, but not least, the Ph.D. School, its members and the Head of the Ph.D. school, Peter Eklund, for having given me all the support a student can possibly have.

Marco Paviotti
Copenhagen, June 13rd, 2016

Contents

Declaration of Authorship iii

Abstract v

Resume vii

Acknowledgements ix

I Overview 1

1 Introduction 3
1.1 Semantics of programming languages 5
1.1.1 Computational Adequacy 6
1.1.2 Domain theory 6
1.2 Criticism of denotational semantics 7
1.3 Axiomatic and synthetic domain theory 8
1.3.1 Synthetic formalisations in type theory 8
1.4 Formalisations of Domain Theory in Type Theory 9
1.5 Step-indexing in the metric spaces 10
1.5.1 Step-indexing 10
1.5.2 Metric Spaces 11
1.5.3 Relational reasoning using the metric spaces 12
1.5.4 Escardó metric space model of PCF 12
1.5.5 Further uses of the metric spaces 13
1.6 Synthetic Guarded Domain Theory 13
1.6.1 The topos of trees model 14
1.6.2 PCF in $\mathbf{Set}^{\omega^{\mathrm{op}}}$ 15
1.6.3 Escardó's metric model and the topos of trees 17
1.7 Contributions of this thesis 18
1.7.1 Recursion in Synthetic Guarded Domain Theory . . . 18
1.7.2 Recursive Types in Synthetic Guarded Domain Theory 19
1.7.3 Applications of synthetic step-indexing to formal verification 20
1.8 Future work 21
1.9 Details of publications 22

II Recursion in Guarded Recursion 23

2 Recursion in Guarded Type Theory 25
2.1 Introduction 25
2.2 Guarded recursion 26
2.2.1 The topos of trees model 28

2.3 PCF . . . 28
2.3.1 Big-step semantics . . . 29
2.3.2 Small-step semantics . . . 30
2.4 Denotational semantics . . . 35
2.4.1 Interpretation . . . 36
2.4.2 Soundness . . . 38
2.5 Computational Adequacy . . . 39
2.5.1 Guarded Dependent Type Theory . . . 39
2.5.2 Logical Relation . . . 40
2.6 The external viewpoint . . . 44
2.7 Discussion and Future Work . . . 45

3 Recursive Types in Synthetic Guarded Domain Theory 47
3.1 Introduction . . . 47
3.1.1 Synthetic guarded domain theory . . . 48
3.1.2 Contributions . . . 49
3.1.3 Related work . . . 49
3.2 Guarded recursion . . . 50
3.2.1 The topos of trees model . . . 51
3.3 FPC . . . 51
3.3.1 Small-step semantics . . . 52
3.3.2 Big-step semantics . . . 54
3.3.3 Examples . . . 55
3.3.4 Equivalence of small-step and big-step semantics . . . 56
3.4 Denotational Semantics . . . 62
3.4.1 Interpretation of types . . . 62
3.4.2 Interpretation of terms . . . 66
3.5 Computational Adequacy . . . 71
3.5.1 Delayed substitutions . . . 71
3.5.2 A logical relation between syntax and semantics . . . 72
3.5.3 Proof of computational adequacy . . . 73
3.6 Extensional Computational Adequacy . . . 82
3.6.1 Universal quantification over clocks . . . 82
3.6.2 Global interpretation of types and terms . . . 83
3.6.3 A weak bisimulation relation for the lifting monad . 83
3.6.4 Relating terms up to extensional equivalence . . . 86
3.6.5 Properties of $\approx_{\Gamma,\tau}$ . . . 86
3.7 Conclusions and Future Work . . . 95

III Applications to step-indexing 97

4 Applications to step-indexing 99
4.1 Introduction . . . 99
4.2 Memory allocation using exceptions . . . 100
4.2.1 Interrupt mechanism . . . 101
4.3 Assembly semantics and logic . . . 103
4.3.1 Semantics . . . 103
4.3.2 Separation Logic for low-level code . . . 104
4.3.3 Memory representation . . . 107
4.4 Semantics and logic for exceptions . . . 108

    - 4.4.1 Semantics for exceptions 108
    - 4.4.2 Specification logic for exceptions 109
  - 4.5 Memory allocation using exceptions 110
    - 4.5.1 The memory datatypes in Coq 110
    - 4.5.2 Specification for the allocator 111
    - 4.5.3 Proof of the specification 112
  - 4.6 Related Work 114
  - 4.7 Conclusions and Future Work 114
  - 4.8 Acknowledgements 115

A Set theory vs Type Theory **117**
    - A.0.1 Set theory 117
    - A.0.2 Type theory 118
    - A.0.3 Relating type theory and set theory 119

Bibliography **121**

List of Figures

1.1.1 The type LA in the model 16
1.1.2 The type $\triangleright LA$ in the model 16

2.0.1 PCF typing rules 28
2.0.2 Step-indexed Big-Step Operational Semantics for PCF 29
2.0.3 Step-Indexed Small Step semantics of PCF. In the rules, k can be 0 or 1. 30
2.14.1 Interpretation of terms 36

3.0.1 Rules for welformed FPC types 52
3.0.2 Rules for welformed FPC contexts 52
3.0.3 Typing rules for FPC terms 53
3.0.4 Reductions of the small-step call-by-name operational semantics. In the last rule, k is either 0 or 1. 53
3.2.1 The big-step operational semantics. In the definitions of Q' only non-empty cases are given, e.g., in the case of $\texttt{unfold}\,M$, $Q'(P, n)$ is defined to be the empty type unless P is of the form $\texttt{fold}\,N$ and n is a successor. 54
3.14.1 Interpretation of FPC types 63
3.16.1 Definition of $\theta_\sigma : \triangleright [\![\sigma]\!] \to [\![\sigma]\!]$ 66
3.16.2 Interpretation of FPC terms 67
3.23.1 The notation $\xi\,[x \leftarrow t]$ means the extension of the delayed substitution ξ with $[x \leftarrow t]$. Rule (3.17) requires x not free in u. Rule (3.19) requires that none of the variables in the codomains of ξ and ξ' appear in the type of u, and that the codomains of ξ and ξ' are independent. 72
3.23.2 The logical relation $\mathcal{R}_\tau : [\![\tau]\!] \times \texttt{Term}_{\texttt{FPC}} \to \mathcal{U}$. 73
3.38.1 The logical relation $\approx_\tau$ is a predicate over denotations of τ of type $[\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$ 86
3.45.1 Definition of the "fill hole" function 89
3.45.2 Typing judgment for contexts 90

4.0.1 Standard Allocation mechanism 101
4.0.2 Allocation mechanism with exceptions 102

To my family

Part I

Overview

Chapter 1

Introduction

Humanity's enthusiasm for complex computer systems has outpaced our ability to safely design them. To build a bridge, a house or a skyscraper, a team of civil engineers has to scrupulously perform all the necessary calculations and take into account all the possible things that can go wrong. However, because computer science still lacks a proper mathematical understanding this practice cannot yet be applied to software engineering.

One manifestation of this problem is our perennial inability to secure software systems against possible failures and leaks of information. To fix this problem we need to be able to specify and prove computer programs correct. To this end, we need mathematically precise languages that allow us to give full specifications of the behaviour of programs and to prove programs correct with respect to their specifications. In other words, we need logics, mathematical models and formal systems equipped for reasoning about computer programs.

Program verification is a discipline of computer science that focuses on finding specification languages and models for programming languages with the ultimate aim of providing tools for proving software correct w.r.t. some specification. However, verifying computer programs with more than, say, a dozen of lines of code, using pen and paper is a daunting and error-prone task. To make sure proofs of specifications mechanically correct we need tools (other computer programs called proof-assistants) to check the proofs against the specifications. Still, verifying even a small code snippet in a proof-assistant is *also* a daunting task and requires a lot of effort. This practice is therefore relegated to the niche of safety critical systems where lives – not to mention millions of dollars – are at stake.

To be able to overcome these limits in computer science we need three things: clear and comprehensible specification languages for real-world programming languages, tools that implement them and, finally, programming languages with abstractions that relate to some good properties in order to filter out as many programming mistakes as possible. For example, when a programmin language is *strongly-typed* all the programs written by the user are guaranteed to be free from typing errors. This is a property of most functional programming languages [Gun92; Win93; Sch86; LSS84]. Central features in functional programming include *recursion* and *recursive types*. The first permits to write all possible computable functions whereas the second allows the user to create new data types.

Semantics of programming languages is a subfield of computer science that aims to specify the behaviour of computer programs within a mathematical theory. In this way it is possible to formally check, by hand or by another computer program, that a program is correct with respect to its specification.

Denotational semantics aims to give formal semantics of programming languages that can be used to derive operational properties about programs. Rather than focusing on the operational behaviour of a program as in *operational semantics*, denotational semantics abstracts away from the particular syntax and computational steps that a program may have and focuses instead on its *mathematical meaning*. This means that types of the language are interpreted as structures called *domains* and programs as particular functions between them. The additional structure is necessary to obtain a sound interpretation. The model obtained can be used to reason about programs by means of a completeness result which we call *computational adequacy*.

The study of giving denotational semantics using domains is called *domain theory*. Denotational semantics are used to formally specify the behaviour of programs so that we can use them to prove properties and to *inspire* new programming languages and new logics to reason about programs that can be implemented as a proof-assistant.

However, procuring the structure described by domain theory to embed featureful programming languages requires a lot of technicalities that are hard to grasp to the common audience. Category theory [ML71] was also used to solve this problem, however, most computer scientists have long criticised it as far too complex.

The complexity of denotational semantics comes from the fact that the mathematical theory upon which the whole framework rests, set theory, lacks the required structure to represent the basic notions of computation. This problem led to *synthetic* variants of domain theory which went under the name of Synthetic Domain Theory (SDT). In other words, SDT is a search for new meta-theories with more suitable properties for programming languages. This is done by taking some constructive variant of set theory and axiomatising the properties of domains inside it.

There is another field of computer science devoted to the study of more appropriate and comfortable foundations: *type theory*. Its central idea is that a typed programming language should be regarded as a logical system (and viceversa) under the slogan "*formulas as types and proofs as programs*". This way it is possible to program and at the same time prove properties using the type system of the language. This idea has wide-ranging implications: some computer scientists now even state that "*Mathematics is a branch of Computer science*" (cit. Robert Harper, Oregon, 2013) due to the possibility that this new theory might be a better tool for the whole realm of mathematics and not only for that of computer science.

An important line of research is the formalisation of programming languages and semantics into type theories or logical frameworks [CH; HHP93]. The goal of this work is to facilitate machine-assisted reasoning about programs.

There exist many formalisations of programming languages and semantics inside type theory [BKV09]. One way to do this, is to make all the definitions explicit within the prover's logic. For example, defining a relation that states when a program *reduces* to another with a computational step, or, defining domain-theoretic construction inside type theory the same way as it is done in set theory. However, reasoning about programs with this kind of semantics is hard as the underlying theory does not understand the meaning of these new definitions.

We would like to formulate these languages *synthetically*, namely, interpreting types of the language as types in the type theory and programs of the language as type-theoretical terms. This way, we could maintain all the benefits of the proofs-as-program paradigm and, moreover, would allows us to reason about programs using native mechanisms of the chosen type theory, for example, of dependent types.

The results provided in thesis here are both type theoretic and synthetic. In particular, we use a type theory with guarded recursion to give a computational adequate model of functional programming languages with recursion and recursive types. The whole development, the operational semantics, the denotational semantics and the proofs, is carried out *entirely inside the type theory*. This means that the type theory might be used in the future to reason about operational properties of programs.

In following sections we explain these points in detail. In particular, we introduce Guarded Dependent Type Theory (gDTT) and explain why it is suitable for doing synthetic domain theory. In particular, we claim that our formulation is much easier to understand when proving properties about programs, and that it potentially allows us to reason about a wide range of much more challenging functional programming languages for which denotational techniques have previously been shown to be impractical. Furthermore, this thesis adds significant value to gDTT by showing another very useful application of it.

As a slightly philosophical side remark, this work also hints that the *meaning of computation* sits much more comfortably into constructive mathematical theories with a notion of *time*.

Warning: some of the definitions used in this chapter are merely used to introduce our work and give the reader some flavour of the ideas underpinning this thesis. They are not meant to be fully rigorous and we defer the reader to the appropriate references for further reading.

1.1 Semantics of programming languages

We start by introducing semantics of programming languages. A natural way of describing a programming language is via *operational semantics*, which are described by a relation $E \Rightarrow E'$ specifying that the expression E makes one computational step to E'. This relation specifies *how* a program computes, thus leading to a more *intensional* description. Another way, would be to define a relation $E \Downarrow v$ stating that E terminates with value v.

However, syntactically E is still different from v. A more extensional notion would say that E and v are *equal*. As a result, reasoning with equality would be much easier than reasoning with arbitrary relations in the meta theory. Perhaps more philosophically, from a mathematical point of view it may be irrelevant to worry about the dynamic aspects of execution or the syntactic differences between programs. Thus, one may concentrate on the *what* and forget about the *how*.

In denotational semantics closed terms or expressions E of type σ are interpreted into a mathematical object $[\![E]\!] \in [\![\sigma]\!]$, called the denotation of E, where $[\![\sigma]\!]$ is a previously defined structured set called *domain*. Closed expressions of type $\sigma \to \tau$ will be interpreted as particular functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$ which are considered equal when they deliver the same result for all arguments. In other words, the meaning of such a program is fully determined by its input/output behaviour.

When a programming language comes endowed with an operational and a denotational semantics there arises the question of how well they fit together.

1.1.1 Computational Adequacy

When reasoning at the syntactic level with operational semantics we may want to ask ourselves if two programs P and Q are *observationally equivalent*, i.e. if they perform the same actions even though they are not exactly the same program. A more formal way of stating this is by taking all possible contexts C with a hole and *testing* if $C[P]$ and $C[Q]$ reduce to the same value. If this is the case it means that no C can tell the difference between P and Q. This is what one would expect for example when we use two different libraries that implement the same interface. However, this way of reasoning is tedious as it involves quantifying over all contexts C, whereas, reasoning about equality using the denotational semantics is much simpler.

Computational adequacy states that if two programs are equal in the model then they are contextually equivalent. This allow us to reason about programs directly in the model.

1.1.2 Domain theory

Denotational semantics was due to Dana Scott in 1969 [Sco69]. He showed that to give semantics of the untyped lambda calculus one first has to solve the equation

$$D \cong D \to D \tag{1.1}$$

Unfortunately, this equation does not have a solution in general as the cardinality of $D \to D$ is always bigger than D. This result is due to Cantor. To overcome this problem Scott defines domains as *complete partial orders with a bottom element* (`Cpo`s) and *continuos functions* between them[1], thus restricting the number of functions in the function space.

Domain theory can also be used to model functional programming languages with explicit recursion at the term level and simple types (PCF). The main problem is to show that all the PCF-denotable terms possess a suitable fixed point. In fact, PCF has a Y combinator computing for every term M of type $\sigma \to \sigma$ its fixed point. Thus, to give semantics of this language we need to find a corresponding mathematical description of this term. Since its operational semantics unfold fixed points by means of the rule $\mathsf{Y}_\sigma\ M \to M(\mathsf{Y}_\sigma\ M)$, to obtain soundness we have to interpret Y such that

$$[\![\mathsf{Y}_\sigma\ M]\!] = [\![M]\!]\,([\![\mathsf{Y}_\sigma\ M]\!]) \tag{1.2}$$

As a matter of fact, for all the continuos endofunctions $f : X \to X$, where X is a `Cpo`s, there exists a fixed-point for f, written $\mathsf{fix}(f)$, defined as

$$\mathsf{fix}(f) \overset{\text{def}}{=\!=} \bigsqcup_{n \in \omega} f^n(\bot) \tag{1.3}$$

---

[1] There exists extensive literature in this matter. Thomas Streicher's book is probably the most accessible one [Str06]

where $\bigsqcup_{n\in\omega} x_n$ is the *least upper bound* of all x_n , in other words, the most precise approximation of the computation. The presence of the fixed point is guaranteed by continuity of all interpreted functions.

Domain theory has been also used to give denotational semantics of a lambda calculus with recursive types (FPC). Recursive types allow the user to create user-defined types, e.g. natural numbers are encoded by the recursive type $\mu X.1 + X$.

Again, the question is whether we can give denotational semantics for this kind of languages. A first naive approach could be to interpret each type as a domain and then proceed by induction on the types. However, in the case of recursive types $\mu\alpha.\tau$ we need to know that the interpretation of the type $\tau[\mu\alpha.\tau]$ is defined. For example, by defining the fixed point case as follows

$$[\![\mu\alpha.\tau]\!]_\rho \overset{\text{def}}{=\!=} [\![\tau]\!]_{\rho[[\![\mu\alpha.\tau]\!]/\alpha]} \tag{1.4}$$

where ρ is a function from type variables to the codomain of the interpretation function. This would imply by Substitution Lemma on types that

$$[\![\mu\alpha.\tau]\!] \cong [\![\tau[\mu\alpha.\tau/\alpha]]\!] \tag{1.5}$$

However, (1.4) is not well-defined as this definition is *circular* : the object we are trying to define is mentioned in the definition. More specifically, induction hypothesis requires a syntactically smaller type than $\mu\alpha.\tau$. However, the mathematics here gets complicated: the interpretation function needs some kind of fixed-point property that in standard set theory does not exists. Smyth and Plotkin [SP77] reformulate the solution to this problem using category theory, resulting in a more abstract representation.

Among functional programming languages that became mainstream are the likes of ML includes general references. The ability to store in the memory values of *any* type. To give denotational semantics of this languages we need to solve an equation similar to (1.1) and more precisely the kind we will illustrate in Section 1.5.1.

1.2 Criticism of denotational semantics

Probably, Scott's original intent was to encode a type for a functional program as a *set* and a program as a *function*. However, Scott realised that he needed more structure than he originally thought, thus leading to an overcomplicated mathematical theory.

In fact, domain theory has been long criticised for being technically involved and for not scaling properly to more interesting languages. At the present day, a lot of work has been done on semantics of languages with simple types, recursive types, polymorphism and general references. However, as we have hinted above for recursive types the technical details involved to even define proper relations to prove computational adequacy are out of reach to most computer scientists. Polymorphism is also tricky to model. More interestingly, at the present day and to the best of our knowledge, we do not know of any *denotational model for general references*[1].

---

[1] By *denotational model* we mean an *adequate* mathematical model where types are interpreted as domains and terms as functions between them.

1.3 Axiomatic and synthetic domain theory

The structure needed at the set theoretical level became too heavy to bear and category theory [ML71] offered the right level of abstraction that made possible to treat huge quantity of details with simple constructions. This led to the quest for a *category of domains* or *axiomatic domain theory* [FP94; Fio96], i.e. a category with enough structure to abstractly give models of programming languages.

On the other hand, Scott's original vision was to have *domains as sets*. In words, take set theory and postulate the existence of some sets with special properties and arbitrary set-theoretical functions. It was the same Scott who pointed out [Sco80] that this idea is inconsistent with classical set theory and that it might be consistent with constructive set theory. As a consequence of this statement, a long line of work kicked off towards a new meta theory called Synthetic Domain Theory [Tay91; Hyl91; Pho91; Ros86; RS99; Sim02; RS04].

The idea of SDT is roughly that computability is built into the logic. As a result, constructions on domains would be set-theoretic with no extra structure and proofs of continuity for free. In other words, intuitionistic higher-order logic along with an axiomatisation of domain theory. The advantage of this approach is that it would be model independent and therefore formalisable in some theorem prover implementing intuitionistic logic.

LCF (Logic of Computable Functions) [Mil72] is a logic to reason about programs and is based on Scott's model [Sco69]. LCF and its descendants constitute a whole set of machine-assisted tools for program verification using higher-order logic. Milner's LCF theorem prover laid the basis for more recent proof-assistants such as HOL and Isabelle [tea88].

1.3.1 Synthetic formalisations in type theory

LCF-based proof assistants have proved very useful for encoding domain theory [Age95; Bar+96; Reg95] with some notable semantic applications [VB08]. On the other hand, since these proof-assistant rely on classical logic we cannot take full advantage of dependent types nor of constructivism.

In this direction the most notable work is the one by Reus [Reu95; Reu96]. He extended the Extended Calculus of Constructions [Luo90], a type theory with an impredicative univers, by adding an axiomatisation of domain theory to it, proving it consistent w.r.t. a realisability model and formalising the whole theory in LEGO [Pol94]. This is done by postulating a new *impredicative* universe and assuming there exists a special set Σ to classify the *semi-decidable* predicates. This has the advantage that we can machine check properties about programs using the logic inside a proof-assistant. Moreover, in this logic continuity proofs are for free.

On the side remark, other work might sound prompted in this direction. Plotkin proposed to use intuitionistic type theory with linear maps, in fact an *intuitionistic linear type theory* [Plo93]. This idea was realised by Møgelberg [Møg06; Møg09]. He encoded a lambda calculus with polymorphism and recursive types into models of polymorphic intuitionistic linear lambda calculus. Despite this work shows some useful direction to look at, it is far from being "a proper" type theoretic formalisation in that all the work is carried out in set-theory.

A common problem with SDT approaches in type theory Basically, the SDT approach is to take an intuitionistic logic, be it intuitionistic set theory or a type theory, and add some axioms that best represent the properties of domain theory.

However, it is clear that a proper synthetic approach to domain theory in type theory, namely, where types are domains and programs are functions in the type theory will never exist. The reason is that in type theory adding an unrestricted fixed-point operator for all types is inconsistent. More precisely, if we added as an axiom that states that for all types X in the constructive universe and terms $f : X \to X$ there exists a fixed point for f, we would end up with an inconsistency. In fact, as the type $X \to X$ is always inhabited (the identity function has this type) we would be able to produce via the fixed-point combinator an inhabitant of every type. For instance, an inhabitant of the empty type.

Moreover, adding axioms to type theory surely does not necessarily prevents the result to be fully constructive, but finding a constructive interpretation of such axioms requires additional effort. On the other hand, we would like to have a formalisation of denotational semantics entirely formulated inside the constructive universe of some type theory.

1.4 Formalisations of Domain Theory in Type Theory

Another approach is to fully encode domain theory giving definitions and proofs of basic results on `Cpos` and continuous functions explicitly within the prover's logic. In other words, replacing set theory with type theory. Due to domain theory relying on classical mathematics this has some problems, for example, sets need to be encoded as there is no native notion of set in type theory. Moreover, one may be tempted to take types and endow them with a bottom element, for instance for a type X, defining our domain as $X + \{\bot\}$. Since, in type theory functions are total and computable, this way would allow to interpret a program as a total function $[\![M]\!] : X + \{\bot\} \to Y + \{\bot\}$. This implies that we could run the function and see whether it returns the bottom element or not. By adequacy theorem if a denotable term gives bottom its syntactic counterpart diverges. This implies decidability of the halting problem.

To overcome this problem, Capretta [Cap05] defined a *lifting monad*, as the greatest solution to the equation

$$LA \cong A + LA \tag{1.6}$$

where the infinite element defines the diverging computation. In this way, a total function cannot decide non-termination. The coinductive lifting monad proved its worth and it has been used by many others to formalise semantics of programming languages in proof-assistants. Benton et. al. [BKV09] attempted to formalise domain theory in Coq. However, in *loc.cit.* they report that "*The constructive nature of our formalization and the coinductive treatment of lifting has both benefits and drawbacks*". In particular, their conclusion is that some constructions such as the smash product of pointed domains are not possible and one has to move to unpointed ones. On the other hand, this effort seems to pay off. As noted in *loc.cit*, dependent types are "*necessary if one wishes to prove theorems like adequacy or do compiler correctness*" moreover, dependent types are convenient for "*working with monads and logical relations*". Finally, an interesting, yet unexplored path, would be to exploit the Curry-Howard isomorphism via the *Extraction* mechanism of Coq to extract runnable code out of operational flavour of the semantics given inside the lifting monad.

This line of work suggests that domain theory is hard to encode in type theory. But, domain theory is not the only suitable theory that ensures the presence of fixed-points. Circularities like the one in (1.1) can be broken introducing well-founded structures, for example natural numbers, which do ensure the presences of fixed-points and isomorphisms. This technique is known as *step-indexing*.

1.5 Step-indexing in the metric spaces

Domain theory has been used to guarantee the presence of fixed-points at the term level as the one in (1.3) and to solve particular kind of domain equation such as the one in (1.1).

Step-indexing is a also a technique to ensure the presence of fixed-points and its abstract counterpart, metric spaces is a setting in which is possible to solve domain equations.

These techniques have been successfully applied to reasoning techniques such as relational reasoning, program logics and denotational models [Esc99; Ahm04; Ahm06; DAB11; SB14; Jun+15].

The price one has to pay is *intensionality*.

1.5.1 Step-indexing

The idea of step-indexing is to artificially provide a means for recursion. In this introduction we have provided a wide variety of cases in which certain solutions to some equations do not exists. The domain-theoretic solution is to solve these equations into a more structured "host theory".

Step-indexing differs from the domain-theoretic practices we introduced so far. The trick is to tweak the definition by adding *natural numbers*, thus solving a similar, but not identical, problem. Experience tells us that this is enough.

For example, in relational reasoning it is customary to define a relation on terms such that it implies contextual equivalence. This relation may have the following shape

$$[\![\sigma]\!] \in \mathcal{P}(\mathbf{V} \times \mathbf{V})$$

where $\mathbf{V}$ is the set of values for a language and $[\![\cdot]\!]$ is recursively defined on the types. In the presence of recursive types a first naive attempt would be to define the interpretation function as follows

$$[\![\mu\alpha.\tau]\!]_\rho = \left\{(\texttt{fold}\, v, \texttt{fold}\, v') \mid (v, v') \in [\![\tau[\mu\alpha.\tau/\alpha]]\!]_\rho\right\}$$

However, as in (1.5), we cannot appeal to the induction hypothesis on types as $\tau[\mu\alpha.\tau/\alpha]$ is syntactically bigger than $\mu\alpha.\tau$. The solution is to break circularity by adding natural numbers, thus by defining the interpretation function as

$$[\![\sigma]\!] \in \mathcal{P}(\mathbb{N} \times \mathbf{V} \times \mathbf{V})$$

and give the interpretation function first by induction on the index and then by induction on the type. This way, when the type does not get smaller, we can use the index.

However, things get more tricky when dealing with general references. Two references are related if, intuitively, the memory they represent is related. In the presence of higher-order store, however, the memory can contain programs as well. Therefore, we need an environment, a *world*, that for every allocated location gives us a relation. This was first noticed by Ahmed [Ahm04]. A first approach is to defined our domain of relations as follows

$$\begin{aligned} \mathcal{T} &\cong \mathcal{W} \to_{\text{mon}} \mathcal{P}(\mathbf{V} \times \mathbf{V}) \\ \mathcal{W} &\cong \mathbf{Loc} \to_{\text{fin}} \mathcal{T} \end{aligned} \tag{1.7}$$

Once again, this definition is not well-defined as the recursive variable $\mathcal{T}$ appears in the negative position. However, the problem here is to define a *domain*, i.e. a set of relations, whereas, in the case of the recursive type the domain was indeed well-defined. This problem is dealt with more easily by abstracting step-indexing using category theory, thus moving the setting where objects come equipped with a notion of distance and maps respecting this structure.

1.5.2 Metric Spaces

The category of metric spaces is a category where objects are pairs $(\mathcal{M}, d)$ where $\mathcal{M}$ is a set and $d : \mathcal{M} \times \mathcal{M} \to [0, \infty]$ is a metric: a function that defines a distance between each pair of elements of a set. A map between two spaces is called *non-expansive* when it does not increase the distance between two given points, i.e.

$$d(f(x), f(y)) \leq d(x, y)$$

Similarly to (1.3), in domain theory, this category provides a way to ensure fixed-points of terms. A map between two spaces is called *contractive* if it decreases the distance between them, i.e. if there exists a constant $c < 1$ such that for all $x, y \in \mathcal{M}$

$$d(f(x), f(y)) \leq c * d(x, y)$$

This map is guaranteed to have a fixed point by Banach's fixed point theorem.

Theorem 1.1 (Banach's Fixed Point). *Let $(\mathcal{M}, d)$ be an* inhabited *and* complete *metric space. If $f : (\mathcal{M}, d) \to (\mathcal{M}, d)$ is contractive then f has a unique fixed point.*

The composition of a non-expansive map with a contractive one yields a contractive map.

Similarly, a functor F is *locally contractive* if it reduces a suitably defined distance on morphisms. If $F : \mathcal{M} \times \mathcal{M}^{\text{op}} \to \mathcal{M}$ is a locally contractive functor and $F(1,1)$ is inhabited then there exists a unique solution X such that $F(X,X) \cong X$.

1.5.3 Relational reasoning using the metric spaces

In relational reasoning, the metric spaces has been widely used by Birkedal [BST09; BST12] to give relational models of languages with recursive types, polymorphism and general references. The equation above (1.7) can be solved in $\mathcal{M}$ as follows:

$$\begin{aligned} \mathcal{T} &\cong \frac{1}{2} \cdot (\mathcal{W} \to_{\text{mon}} \mathcal{P}(\mathbb{N} \times \mathbf{V} \times \mathbf{V})) \\ \mathcal{W} &\cong \mathbf{Loc} \to_{\text{fin}} \mathcal{T} \end{aligned} \tag{1.8}$$

where $\frac{1}{2} \cdot -$ is the locally contractive functor, which takes a space and divides the distances between points by two, $\mathcal{P}(\mathbf{V} \times \mathbf{V})$ is turned into a metric space by adding a natural number to each pair of values so that we can equip the set with a suitable metric. The reader has to note that this equation is different from the original one. In order to be usable one has to find suitable functions to work with this new type construction.

1.5.4 Escardó metric space model of PCF

The metric spaces have been used by Escardó [Esc99] to give an adequate model of PCF using the category of metric spaces. In particular, he used the category of *complete bounded ultrametric* spaces (*CBUlt*). In this category every object $\mathcal{M}$ is a metric space where the distance between two points is always of the form r^n for some n and for some chosen non-trivial constant r for which the limit $\lim_{n \to \infty} r^n$ is 0. Moreover, it is *ultrametric* since every distance respect the strong triangle inequality, i.e. for all $x, y, z \in M$, $d(x,z) \leq \max(d(x,y), d(y,z))$.

He encoded the fixed-point combinator of PCF using Banach's fixed point theorem. Roughly, in its development every PCF-denotable map $[\![M]\!] : [\![\sigma]\!] \to [\![\sigma]\!]$ is non-expansive. The trick is to find a contractive map δ, that can be composed with $[\![M]\!]$.

To do this, he defines a *metric lifting* on sets. More precisely, for a set A, he defines a functor $L : \mathbf{Set} \to \text{CBUlt}$ as

$$LA = (A \times \mathbb{N}) \cup \{\infty\} \tag{1.9}$$

where a computation LA either terminates with a finite number of steps yielding a value or diverges (∞). The object LA is a metric space endowed with a distance between points, defined as :

$$d(\infty,\infty) = (d(a,k),(a,k)) = 0 \qquad d((a,k),\infty) = d(\infty,(a,k)) = r^k$$
$$d((a,k),(b,l)) = r^{\min(k,l)} \quad \text{if } a \neq b \text{ or } k \neq l$$

For each set A, we can define the "delay" operator $\delta_A : LA \to LA$ and the unit map $\eta_A : A \to LA$ as follows

$$\delta_A(a,n) = (a,n+1) \qquad \delta_A(\infty) = \infty \qquad \eta_A(a) = (a,0) \tag{1.10}$$

The δ increases the number of steps of a terminating computation by one and leaves the diverging computation divergent, whereas the η map takes a value of type A and gives a computation that terminates in zero steps with the same value.

The metric on LA is uniquely determined by the equations

$$d(\infty,\infty) = d(\eta_A(a),\eta_A(a)) = 0 \qquad d(\eta_A(a),\eta_A(b)) = 1 \text{ if } a \neq b \tag{1.11}$$

$$d(\eta_A(a),\delta_A(x)) = d(\delta_A(x),\eta_A(a)) = 1 \qquad d(\delta_A(x),\delta_A(y)) = r * d(x,y) \tag{1.12}$$

1.5.5 Further uses of the metric spaces

Benton et al. [Ben+10] encoded the metric spaces in Coq to give denotational semantics of languages with recursive types.

The metric spaces are also, perhaps quite surprisingly, connected with coinduction and productive functions. They model Nakano's lambda calculus [Nak00; Bir+12], a calculus for ensuring productivity of coinductive types. This is achieved by a unary modality on types pronounced "later".

The metric spaces can be generalised even further using the category $\mathcal{S}$ of the topos of trees, the category of presheaves over ω, the first infinite ordinal. More precisely, the category of complete bisected ultrametric spaces is a full subcategory of the topos of trees. This was firstly discovered by Birkedal et al. [Bir+12]. We will give a glance of this fact in Section 1.6.3 after having introduced the topos of trees in the next section, though, an accessible presentation of this fact can be found in Bizjak's Ph.D. thesis [Biz16, Section 1.2].

The benefit is that $\mathcal{S}$ is a topos: it has a internal logic and it is a model for Guarded Dependent Type Theory (gDTT) [Biz+16].

1.6 Synthetic Guarded Domain Theory

Guarded dependent Type Theory [Bir+12; BM13; Møg14; Biz+16] is a type theory with a unary modality on types $\triangleright$, pronounced "later" and inspired by Nakano's lambda calculus. If X is a type then $\triangleright X$ is the type of elements available only one step from now. Moreover, if X is a type there is always a map $\mathsf{next} : X \to \triangleright X$. Intuitively, if an element is available now then it is also available later. Furthermore, gDTT is characterised by a restricted fixed point operator on all types X,

$$\mathsf{fix} : (\triangleright X \to X) \to X$$

which permits to construct fixed-points for every term as long as the recursive call is guarded by next.

Finally, we can solve domain equations as the one above as long as the $\triangleright$ operator guards the recursive variable. For example, the domain of world-indexed relations in (1.7) can be solved alternatively as follows:

$$\mathcal{T} \cong \triangleright((\mathbf{Loc} \to_{\text{fin}} \mathcal{T}) \to_{\text{mon}} \mathcal{P}(\mathbf{V} \times \mathbf{V}))$$

Since the $\triangleright$ operator is guarding the recursive variable this definition is well-defined. In previous work, Birkedal et al. [Bir+12] used this equation to give relational models of languages with recursive types, polymorphism and general references. They showed this using the proof-irrelevant universe of gDTT.

Guarded recursive types are very useful also for checking productivity of definitions. In fact, we can define a guarded recursive variant of the coinductive streams as

$$\mathsf{Str} \overset{\text{def}}{=\!=} A \times \triangleright \mathsf{Str}$$

The type discipline here ensures that every element defined under this domain is a *productive* stream.

However, guarded recursive types are stricter than coinductive types – as noted by Atkey and McBride [AM13] – in that they do not allow to look beyond the next step. In other words, once we obtain an element of type $\triangleright$ we cannot access its content even though the term under consideration was still meeting the productiveness conditions. To solve this issue gDTT employs clock variables originally pioneered by Atkey and McBride [AM13]. This allows for a controlled removal of the $\triangleright$ operator. The easiest way to see this is probably by analogy with intuitionistic modal logic IS4 [BP] and it is in fact used for productive programming in the lambda calculus [Clo+15].

On a side remark, we cannot really say gDTT is a fully-fledged type theory as a proper strongly normalising operational semantics does not exists yet. However, an early prototypical work [Bir+16] suggests that operational semantics are to be sought in a combination of guarded recursion and cubical type theory [Coh+15].

To understand how guarded type theory is a synthetic version of step-indexing it might be useful to see its model, the topos $\mathcal{S}$ of trees, and see how Escardó's model fits in there.

1.6.1 The topos of trees model

The category of the topos of trees is the category of presheaves over ω, the first infinite ordinal. An object X in this category is a functor $X : \omega^{\mathrm{op}} \to \mathbf{Set}$. More concretely, X is a ω-indexed chain of sets along with restriction maps r_n^X for $n \in \omega$. Graphically an object looks as follows

$$\begin{tikzcd}
X(1) & X(2) \arrow[l, "r_1^X"'] & X(3)\cdots \arrow[l, "r_3^X"'] & X(n)\cdots \arrow[l, "r_{n-1}^X"']
\end{tikzcd}$$

Given two objects X and Y a map $f : X \to Y$ is natural transformation between X and Y.

The $\triangleright$ operator is interpreted as an endofunctor in $\mathcal{S}$ where $\triangleright X(1) = 1$ and $\triangleright X(n) = X(n-1)$ for $n > 1$. The action on morphism takes a natural transformation f and for the first stage gives the unique mapping into the terminal object 1 and for $n > 1$ returns f_{n-1}. Graphically, $\triangleright X$ has the following shape

$$\begin{tikzcd}
1 & X(1) \arrow[l, "!"'] & X(2)\cdots \arrow[l, "r_1"'] & X(n-1)\cdots \arrow[l, "r_n"']
\end{tikzcd}$$

As hinted previously, there always exists a map $\mathsf{next} : X \to \triangleright X$. This is modelled by using the restriction maps. Using terminology from the metric spaces, a morphism $f : X \to Y$ is contractive if there exists a morphism $g : \triangleright X \to Y$ such that $f = g \circ \mathsf{next}_X$.

The fixed-point is a family of morphisms $\mathsf{fix}_X : (\triangleright X \to X) \to X$, indexed by all objects X such that if $f : X \to X$ is contractive, witnessed by a

$g : \triangleright X \to X$ then $\mathsf{fix}_X(\hat{g})$ is unique map $1 \to X$ such that $f \circ \mathsf{fix}_X(\hat{g}) = \mathsf{fix}_X(\hat{g})$ where $\hat{g}$ is the exponential transpose.

Fixed-points of endofunctors also exists under certain properties. Reusing some nomeclature from the metric spaces, these are the endofunctors that are *strong* and *locally contractive* [Bir+12]. For the reader, it suffices to observe that every functor constructed out of the usual type theoretical operators ($\times$, $+$ and $\to$) where the recursive variable is guarded by the $\triangleright$ functor is locally contractive and therefore admits a *unique fixed-point*.

To illustrate this relation, in the next section, we reformulate Escardó's metric model into the topos of trees and show we can give a more intensional version of the lifting monad in (1.6). Our version is very similar to Escardó's metric lifting (1.9).

1.6.2 PCF in $\mathbf{Set}^{\omega^{\mathrm{op}}}$

The guarded recursive lifting monad is defined as the fixed-point of a functor F_A defined as $F_A X = A + \triangleright X$ for any object A. By repeatedly applying F it is easy to see that the fixed-point indeed exists. We do this on a generic object X. Since F admits a unique fixed-point it does not matter what X is.

$$\begin{aligned}
(FX)(n) &= A(n) + X(n-1)\\
(FFX)(n) &= A(n) + A(n-1) + X(n-2)\\
(FFFX)(n) &= A(n) + A(n-1) + A(n-2) + X(n-3)\\
&\vdots\\
(\mu X.FX)(n) &= A(n) + A(n-1) + A(n-2) + \cdots A(1) + 1
\end{aligned} \tag{1.13}$$

The lifting functor $L : \mathcal{S} \to \mathcal{S}$ is defined to be the fixed-point of F, namely $LA = \mu X.F_A X$. Thus, the application of L at the nth component has the following shape:

$$LA(n) = A(n) + A(n-1) + \cdots + A(1) + 1$$

the intuition behind this construction is that if we look at the nth approximation of a computation we can observe a computations that produce a value in less or exactly n steps or diverge. The restriction maps are depicted in Figure 1.1.1. Note also that if we apply the $\triangleright$ operator to the object LA we obtain the object in Figure 1.1.2. If we apply the functor $A + -$ back to $\triangleright LA$ we obtain exactly LA in Figure 1.1.1, hence

$$LA \cong A + \triangleright LA \tag{1.14}$$

Moreover, when A is constant, the reader can easily check that the following isomorphism holds

$$LA(n) \cong \{0, 1, ..., n-1\} \times A + 1$$

whose meaning is very similar to (1.9) except that here at stage n we can only observe computations that terminate in $n-1$ steps. The isomorphism maps a value $v \in A(n)$ as $(0, v)$ and a value $v \in A(1)$ to $(n-1, v)$ and the unit in 1 to the identity.

$$\begin{tikzcd}
 & & 1 \arrow[ddl] \\
 & & + \\
 & 1 \arrow[ddl] & A(1) \arrow[l] \\
 & + & + \\
1 & A(1) \arrow[l] & A(2) \arrow[l] \\
+ & + & + \\
A(1) & A(2) \arrow[l] & A(3) \arrow[l]
\end{tikzcd}$$

FIGURE 1.1.1: The type LA in the model

$$\begin{tikzcd}
 & & 1 \arrow[ddl] \\
 & & + \\
 & 1 \arrow[ddl] & A(1) \arrow[l] \\
 & + & + \\
1 & A(1) \arrow[l] & A(2) \arrow[l]
\end{tikzcd}$$

FIGURE 1.1.2: The type $\triangleright LA$ in the model

Again, the object we just constructed is different from the coinductive lifting monad in (1.6). In fact, in Figure 1.1.1 if we look at each single stage we find an increasing amount of information about the computation. In the coinductive lifting monad this information is constant.

When A is constant, it is also easy to see by looking at the restriction maps and using the isomorphism above that the next map on objects LA is defined as

$$\mathsf{next}_n(x) = \begin{cases} \bot & x = (n, v) \\ (i, v) & x = (i, v) \text{ and } i < n \\ \bot & x = \bot \end{cases}$$

Thus, if we have a computation at stage n, then after one step, if the computation is divergent it remains divergent. If the computation computes for exactly n steps then it becomes divergent at the stage $n-1$ and if it consumes less than n steps it is mapped to the identity.

Delay and Tick operations The isomorphism in (1.14) induces a $\triangleright$-Algebra, i.e. a map of type $\triangleright LA \to LA$. This is essential in our work since it provides exactly a suitable map to feed to the guarded fixed-point combinator. Abstractly, to construct such a map, we just use the injection map $\triangleright LA \to A + \triangleright LA$ and then use the isomorphism map. We call this map θ.

Concretely, when A is constant, θ is defined as

$$(\theta_{LA})_n(x) = \begin{cases} (i+1, v) & x = (i, v) \\ \bot & x = \bot \end{cases}$$

of course, we could have defined differently. However, abstractly θ is factorised by the right injection into the sum type and then by using the isomorphism induced by L. We can now define a map $\delta : LA \to LA$ that delays a computation as $\delta = \theta \circ \mathsf{next}$. By definition of next and θ this map behaves as follows

$$(\delta_{LA})_n(x) = \begin{cases} \bot & x = (n, v) \\ (i+1, v) & x = (i, v) \text{ and } i < n \\ \bot & x = \bot \end{cases} \tag{1.15}$$

Intuitively, the delay operation takes a computation and delays it by one step. But, the type LA of a computation at stage n allows to look at most at n computational steps. So if a computation takes exactly n steps, the delayed computation diverges.

The intuitions on the guarded lifting monad underpin most of the contributions of this thesis. In particular, has been used in Chapter 2 and Chapter 3.

1.6.3 Escardó's metric model and the topos of trees

The category of complete *bisected* ultrametric spaces (*BiCBUlt*) is a full subcategory of complete *bounded* ultrametric spaces (*CBUlt*). Escardó's model can be reformulated by fixing the constant r with $1/2$, thus getting a metric space where all non-zero distances are of the form 2^{-n} for some n which are the bisected ones.

This definition gives raise to an equivalence relation

$$x \overset{n}{=} y \leftrightarrow d(x, y) \leq 2^{-n}$$

Every object $(\mathcal{M}, d)$ in *BiCBUlt* can be decomposed into a sequence of approximations to form an object in the topos of trees. More specifically, the object in $\mathcal{S}$ at stage n is obtained by quotienting the underlying set $\mathcal{M}$ with $\overset{n}{=}$. Graphically,

$$\begin{tikzcd}
(\mathcal{M}/\overset{1}{=}) & (\mathcal{M}/\overset{2}{=}) \arrow[l, "r_1"'] & (\mathcal{M}/\overset{3}{=})\cdots \arrow[l, "r_3"'] & (\mathcal{M}/\overset{n}{=})\cdots \arrow[l, "r_{n-1}"']
\end{tikzcd}$$

with the identity maps as restriction maps. The category *BiCBUlt*is co-reflexive in the category $\mathbf{PSh}(\omega)$. This means we can also turn an object in the topos of trees into an object in *BiCBUlt*. Given an object X in $\mathcal{S}$ the corresponding metric space is obtained by taking the limit of the presheaf $\lim X_n$ as the underlying set and defining the distance as follows

$$d(x, y) = \bigsqcap\{2^{-n} \mid \forall j < n. \pi_i(x) = \pi_i(y)\}$$

where π_i are the projections from the limit to the ith stage of the presheaf.

The model of PCF we defined in the topos of trees is related to Escardó's one on the metric spaces. One way to see this is by looking at the definition of the guarded recursive lifting monad (1.14). This is an object in $\mathcal{S}$. Since *BiCBUlt* is co-reflexive in $\mathcal{S}$, we can turn the object LA into a metric space by taking its limit. The same way we can turn the delay and unit maps into non-expansive maps in *BiCBUlt*. One can check that the equational laws given by Escardó's (1.11) still hold.

1.7 Contributions of this thesis

Our main contribution is a formalisation of recursion and recursive types in *synthetic guarded domain theory* (SGDT) under the slogan

"Recursion in Guarded Recursion"

More precisely, we give denotational semantics of PCF and FPC using gDTT as a synthetic meta theory in which to formulate the operational semantics, the denotational model and the proofs of adequacy.

For the reasons explained earlier, in order to do synthetic domain theory using a type theory, one would need to extend the type theory itself one way or another. We use a type theory with guarded recursion to do **Guarded** Domain Theory. Our opinion is that guarded recursion is a more natural extension as it has many other useful applications (e.g. checking productivity of coinductive definitions and relational reasoning).

In other words, we turn recursive constructions into guarded recursive ones, be they on terms or types. This is consistent, since the guarded recursive variant of the fixed-point combinator in gDTT is a terminating program. Moreover, the type theory ensures at the type level that only *contractive* functions are fed to the fixed-point combinator. This is similar to SDT in which proofs of continuity were for free.

This means that the main constructions make use of guarded recursive types and thus they turn out to be more *intensional* than the state-of-the-art formulations. To solve this problem, in Chapter 3, we lift the model using a logical relation and proving the extensional adequacy theorem.

1.7.1 Recursion in Synthetic Guarded Domain Theory

In Chapter 2 we give a computationally adequate model of PCF. All the development is carried out inside the type theory with guarded recursion and therefore also the adequacy theorem.

As a result we have to take extra care when formulating the operational semantics. The big-step semantics is defined as an inductive data type inside the type theory. For a term M an natural number k and a value v, $M \Downarrow^k v$ has to be read as "the term M reduces in k steps to a value v". The role of the natural number k is to explicitly count the fixed-point unfoldings. This is necessary to state the adequacy theorem precisely as will see shortly. The case of the fixed-point combinator is the most crucial as here is where we count the steps

$$\mathsf{Y}_\sigma \ M \Downarrow^{k+1} v \overset{\text{def}}{=\!=} \triangleright(M(\mathsf{Y}_\sigma \ M) \Downarrow^k v) \tag{1.16}$$

Thus unfolding of the fixed-point operator consumes one step. We also synchronise the explicit steps with the abstract notion of time in the type theory. This will be clear in a moment. First we define the denotational semantics using the guarded recursive lifting monad in (1.14).

The challenge here is to find a fix-point in the model that represents the fix-point combinator of PCF. In general, a map $M : X \to X$ does not have one. On the other hand, a map $F : \triangleright X \to X$ does. We therefore, first turn the denotation of M into a map $f : \triangleright [\![\sigma]\!] \to [\![\sigma]\!]$ and then we interpret $\mathsf{Y}_\sigma \ M$ into the fix-point of f. The interpretation satisfies a more intensional version than (1.2), namely, one operational step of unfolding correspond to one step in the model as follows

$$[\![\mathsf{Y}_\sigma \ M]\!] = \delta_\sigma([\![M(\mathsf{Y}_\sigma \ M)]\!]) \tag{1.17}$$

Both the model construction and the proof of adequacy are carried out entirely within the type theory. The adequacy theorem is formulated as follows.

Theorem 1.2. *For all well-typed terms M with ground type, $M \Downarrow^k v$ iff $[\![M]\!] = \delta_\sigma^k \, [\![v]\!]$.*

Note that this theorem lives inside the type theory as well. The intensional nature of the denotational semantics forces us to take extra care about the operational semantics. In the semantics, $[\![\mathsf{Y}_\sigma \ (\lambda x.x)]\!]$ is a diverging computation and therefore in the model is always the bottom element. However, also a computation $\delta^{42} \, [\![v]\!]$ at all the stages $n < 42$ is bottom. Intuitively, because there are not enough approximations to see termination. The implication in the adequacy theorem takes into account the notion of time of the type theory. Therefore, also the type $\mathsf{Y}_\sigma \ (\lambda x.x) \Downarrow^{42} v$ must be inhabited for all the stages $n < 42$ in the model. This is done by defining the operational semantics as above, so that if we unfold a computation for 42 times and it does compute for enough steps, for a diverging computation we actually get the type $\triangleright^{42} 0$ which is inhabited for the first 42 approximations as wanted.

Moreover, it is crucial that all the development is carried out inside the constructive universe of the type theory as a formulation of the operational semantics inside the internal logic of the topos $\mathcal{S}$ would not work. In particular, the proof-irrelevant nature of the logic makes the existential to commute with the $\triangleright$ operator. This would make the fixed-point of the identity a terminating program in the operational semantics. This statement can be shown using guarded recursion.

1.7.2 Recursive Types in Synthetic Guarded Domain Theory

In Chapter 3 we give a denotational model of FPC in guarded type theory. In the same style as with PCF we formulate the operational semantics, the denotational semantics and the proofs entirely inside the type theory.

As in PCF, we formulate the operational semantics, the denotational semantics and the proof of adequacy entirely in the type theory. We define an inductive type of the form $M \Downarrow^k v$ to be read as M reduces in k steps to v. For FPC, we count unfolding/folding operations. To make the adequacy theorem to work, we synchronise the explicit step counting with the abstract notion of time in the type theory such that

$$\texttt{unfold}\,(\texttt{fold}\,M) \Downarrow^{k+1} v = \triangleright(M \Downarrow^{k} v) \tag{1.18}$$

The novel character of this work is the interpretation of the recursive types and the definition of the logical relation for proving adequacy. In particular, we interpret the recursive types making use of guarded recursion thus ensuring, almost straigthforwardly that the definition of the interpretation is well-defined

$$[\![\mu\alpha.\tau]\!]_\rho = \triangleright\,[\![\tau[\mu\alpha.\tau/\alpha]]\!]_\rho \tag{1.19}$$

Note that this definition is more intensional than (1.5). In particular, this means that in the meta theory we will need one computational step to look at the information contained in the type. In particular, computational adequacy says that if the denotations of two terms are equal then they converge in the same number of steps to the same value and thus programs that produce the same result but implementing a different algorithm will be different in the model. An *extensional* version of the adequacy result does not take into account the number of steps stating that if the denotations of two terms are equal then the reduce to the same value. In order to recover the extensional version of the adequacy result we first define a relation $\approx_\sigma : [\![\sigma]\!] \times [\![\sigma]\!] \to \mathcal{U}$ that relates objects in the type theory that denote programs that produce the same value. In the case of the recursive type,for x and y of type $[\![\mu\alpha.\tau]\!]$, the relation is defined such that

$$\theta(x) \approx_{\mu\alpha.\tau} \theta(y) = \triangleright(x \approx_{\tau[\mu\alpha.\tau/\alpha]} y)$$

If two computations are related and both produce a computational step, they are related only *later*. Thus, we need to remove the $\triangleright$ operator. To this end, gDTT employs Atkey and McBride's clock variables. Intuitively, they correspond to Clouston's box modality [Clo+15] and allows to make all the data available at once. The relation $\approx^{\mathrm{gl}}_\sigma : [\![\sigma]\!]^{\mathrm{gl}} \times [\![\sigma]\!]^{\mathrm{gl}} \to \mathcal{U}$ is obtained lifting the relation to its *global view points* using the above mentioned clock variables. This allows us to prove that if two computations are ticking then after one tick they are still related now.

1.7.3 Applications of synthetic step-indexing to formal verification

Hand-crafted low-level code will always be at the bottom layer of the operating system as it is needed to deal with the lowest layers of the computer system.

At the present time, the most faithful mechanisation of the assembly language is the one by Jensen et al. [JBK13]. They encoded syntax and semantics of the assembly x86 in a way that code can be extracted and run in the actual machine. Moreover, they devised an unstructured specification logic based on separation logic, step-indexing and higher-order frame rules.

Exceptions in the machine are generated by errors, e.g. division by zero, unauthorised memory access and so on. An operating system uses exceptions to catch run-time errors and handle them so that the machine can continue to execute other tasks. To ultimately verify the lowest layer of an operating system we have to be able to reason about exceptions as well.

This means that we need to be able to specify and prove correct programs that fault.

To do this we extend previous work to deal with exceptions. Exceptions in low-level architectures are implemented as synchronous interrupts: upon the execution of a faulty instruction the processor jumps to a piece of code that handles the error. In Chapter 4, we study synchronous interrupts and show their usefulness by implementing a memory allocator. This shows that it is indeed possible to write positive specifications of programs that fault. All of our results are mechanised in the interactive proof assistant Coq.

1.8 Future work

In previous sections we introduced guarded type theory as a type theory for doing denotational semantics of programming languages with recursion. However, there are many other computational aspects and features that ought to be reformulated in type theory. Among others, (co)inductive types, polymorphism and general references.

Languages with *local store* are modelled in categories of *presheaves* [PP02]. These models have to be encoded in the type theory beforehand – just as it was the case for domain theory [BKV09; Ben+10]. On the other hand when higher-order store comes into play one needs a solution to a recursive domain equation [BST09; BST12] that in set theory or conventional type theories does not exist. This time because of a circular dependency between the worlds – maps from location to types – and the types. This problem is very well summarised in Bizjak's Ph.D. thesis and tutorial notes [Biz16; BB16]. Birkedal et al. [Bir+12] use a synthetic form of step indexing – namely guarded recursive types – to break this circularity, thus being able to give *syntactic models* of languages with polymorphism, recursive types and general references.

Giving denotational semantics for general references would require the same techniques, but doing this in type theory would be a daunting task. On the other hand, guarded type theory can be once again used as a synthetic theory for giving denotational semantics – this time – of general references. This is because step-indexing models – disguised as presheaf models – are models of guarded type theory just as well.

In Chapter 4 we describe an extension of the assembly x86 formalisation in Coq. One motivation for doing this was to preliminary explore the possibility of formalising asynchronous interrupts, thus being able to verify schedulers and device drivers. However, since asynchronous interrupts are a form of concurrency, they will have to be handled using some form of concurrent separation logic. Moreover, interrupts handlers are shared-memory processes; they share the CPU flags and registers and also the stack. This suggest that we need a separation logic for shared memory concurrency. However, the only modular program logic to address this issues we know of is iCAP [SB14]. This logic, however, is defined using guarded recursive types. Therefore, this work would require guarded type theory to be encoded in Coq beforehand or an implementation of guarded type theory.

1.9 Details of publications

This dissertation consists of three papers, the first two of these have been presented at peer-reviewed conferences or accepted for publication. The third one is currently under review.

Each publication forms a chapter of this thesis. However, what is presented here is an extension of the published papers with the missing proofs.

Therefore, each chapter can be read independently from the others even though it might be easier to start with Chapter 2 before reading Chapter 3. Chapter 4 can be entirely read on its own.

What follows is a description of the author contributions for each of the papers.

- **Denotational semantics of PCF in Guarded Type Theory**. Marco Paviotti, Rasmus E. Møgelberg and Lars Birkedal. In *Electronic Notes in Theoretical Computer Science*, MFPS 2015.

  The extended version with all the proofs is contained Chapter 2 of this thesis and is summarised in Section 1.7.1.

  This work was carried out during my visit at Aarhus University. I carried out all the research under the supervision of Rasmus and Lars. I also availed myself of very helpful discussions with the members of the Logic and Semantics group at Aarhus University. I wrote the technical part of the paper, whereas Lars and Rasmus wrote introduction and conclusions and helped finalising the published version.

- **Denotational semantics of recursive types in Synthetic Guarded Domain Theory**. Marco Paviotti, Rasmus E. Møgelberg. In *Proceedings of Logic in Computer Science*, LICS 2016.

  The extended version with all the proofs is Chapter 3 of this thesis and is summarised in Section 1.7.2

  I carried out all the research behind this paper under the helpful supervision of Rasmus. I wrote the technical part of the paper, whereas Rasmus wrote introduction and conclusions and helped finalising the published version.

- **Verifying Exceptions for Low-level code with Separation Logic**. Marco Paviotti and Jesper Bengtson. Submitted to *Journal of Logical and Algebraic Methods in Programming*.

  This is contained in Chapter 4 and is summarised in Section 1.7.3.

  I carried out all the research and wrote most of the paper. Jesper wrote the introduction, the related work and conclusions.

Part II

Recursion in Guarded Recursion

Chapter 2

A Model of PCF in Guarded Type Theory

Marco Paviotti, Rasmus Ejlers Møgelberg
and Lars Birkedal

Abstract. Guarded recursion is a form of recursion where recursive calls are guarded by delay modalities. Previous work has shown how guarded recursion is useful for constructing logics for reasoning about programming languages with advanced features, as well as for constructing and reasoning about elements of coinductive types. In this paper we investigate how type theory with guarded recursion can be used as a metalanguage for denotational semantics useful both for constructing models and for proving properties of these. We do this by constructing a fairly intensional model of PCF and proving it computationally adequate. The model construction is related to Escardo's metric model for PCF, but here everything is carried out entirely in type theory with guarded recursion, including the formulation of the operational semantics, the model construction and the proof of adequacy.

2.1 Introduction

Variations of type theory with guarded recursive types and guarded recursively defined predicates have proved useful for giving abstract accounts of operationally-based step-indexed models of programming languages with features that are challenging to model, such as recursive types and general references [App+07; Bir+12], countable nondeterminism [BBM14], and concurrency [SB14]. Following observations of Nakano [Nak00] and Atkey and McBride [AM13], guarded type theory also offers an attractive type-based approach to (1) ensuring productivity of definitions of elements of coinductive types [Møg14], and (2) proving properties of elements of coinductive types [Biz+16]. One of the key features of guarded type theory is a modality on types, denoted $\triangleright$ and pronounced 'later'. This modality is used to guard recursive definitions and the intuition is that elements of type $\triangleright A$ are elements of A only available one time step from now.

In this paper, we initiate an exploration of the use of guarded type theory for *denotational* semantics and use it to further test guarded type theory. More specifically, we present a model of PCF in guarded dependent type theory. To do so we, of course, need a way to represent possibly diverging computations in type theory. Here we follow earlier work of Escardo [Esc99] and Capretta [Cap05] and use a lifting monad L, which allows us to represent a possibly diverging computation of type X by a function into $L(X)$. In Capretta's work, L is defined using coinductive types. Here, instead, we use a guarded recursive type to define L. Using this approach we get a fairly intensional model of PCF which, intuitively keeps track of the number of computation steps, similar to [Esc99]. We show this formally by proving that the denotational model is adequate with respect to a step-counting operational semantics. The definition of this step-counting operational semantics is delicate — to be able to show adequacy the steps in the operational semantics have to correspond to the abstract notion of time-steps used in the guarded type theory via the $\triangleright$ operator. Our adequacy result is related to one given by Escardo in [Esc99]. To show adequacy, we define the operational semantics in guarded type theory and also define a logical relation *in* guarded type theory to relate the operational and denotational semantics. To carry out the logical relations proof, we make crucial use of some novel features of guarded dependent type theory recently proposed in [Biz+16], which, intuitively, allow us to reason now about elements that are only available later.

The adequacy result of this paper may be seen as a version of Plotkin's classic result from domain theory [Plo77] set in guarded type theory. There has been work to formalise domain theory in Coq [BKV09], however, this is difficult due to the use of classical mathematics. In fact, [BKV09] uses a coinductively defined lifting monad similar to that of Capretta [Cap05]. We believe that guarded type theory is more suitable for encoding in proof assistants such as Coq or Agda, and thus this work can be seen as a step towards enabling the use of the models for formal reasoning.

The remainder of the paper is organized as follows. In Section 2.2 we recall the core parts of guarded dependent type theory and the model thereof in the topos of trees [Bir+12; Biz+16]. Then we define a step-counting operational semantics of PCF in Section 2.3 and the denotational semantics is defined in Section 2.4. We prove adequacy in Section 2.5. In Section 2.6 we use the topos of trees model of the guarded type theory to summarize briefly what the results proved in guarded type theory mean externally, in standard set theory. Finally, we conclude and discuss future work in Section 2.7.

2.2 Guarded recursion

In this paper we work in a type theory with dependent types, natural numbers, inductive types and guarded recursion. The presentation of the paper will be informal, but the results of the paper can be formalised in gDTT as presented in [Biz+16]. We start by recalling the core of this type theory (as described in [Bir+12]), introducing further constructions later on as needed.

A guarded recursive definition is a recursive definition where the recursive calls are guarded by time steps. The time steps are introduced via a type modality $\triangleright$ pronounced 'later'. If A is a type then $\triangleright A$ is the type of elements of A available only one time step from now. The type constructor $\triangleright$ is an applicative functor in the sense of [MP08], which means that there is a term $\mathsf{next}\colon A \to \triangleright A$ freezing an element of A so that it can be used one time step from now, and a 'later application' $\circledast\colon \triangleright(A \to B) \to \triangleright A \to \triangleright B$ written infix, satisfying

$$\mathsf{next}(f) \circledast \mathsf{next}(t) = \mathsf{next}(f(t)) \tag{2.1}$$

among other axioms (see also [BM13]). In particular, $\triangleright$ extends to a functor mapping $f\colon A \to B$ to $\lambda x\colon \triangleright A.\, \mathsf{next}(f) \circledast x$.

Recursion on the level of terms is given by a fixed point operator $\mathsf{fix}\colon (\triangleright A \to A) \to A$ satisfying

$$f(\mathsf{next}(\mathsf{fix}(f))) = \mathsf{fix}(f) \tag{2.2}$$

Intuitively, fix can compute the fixed point of any recursive definition, as long as that definition will only look at its argument later. This fixed point combinator is particularly useful in connection with guarded recursive types, i.e., types where the recursion variable occurs only guarded under a $\triangleright$ as, e.g., in the type of guarded streams:

$$\mathsf{Str}^g_A \equiv A \times \triangleright \mathsf{Str}^g_A$$

The cons operation cons for Str^g_A has type $A \to \triangleright \mathsf{Str}^g_A \to \mathsf{Str}^g_A$. Hence, we can define, e.g., constant streams as $\mathsf{constant}\; a = \mathsf{fix}(\lambda xs\colon \triangleright \mathsf{Str}^g_A\, .\, \mathsf{cons}\; a\, xs)$.

Guarded recursive types can be constructed using universes and fix as we now describe [BM13]. We shall assume a universe type $\mathcal{U}$ closed under both binary and dependent sums and products as usual, and containing a type of natural numbers. We write $\widehat{\mathbb{N}}$ for the code of natural numbers satisfying $\mathsf{El}(\widehat{\mathbb{N}}) \equiv \mathbb{N}$ and likewise $\widehat{\times}$ for the code of binary products satisfying $\mathsf{El}(A \mathbin{\widehat{\times}} B) \equiv \mathsf{El}(A) \times \mathsf{El}(B)$. The universe is also closed under $\triangleright$ in the sense that there exists an $\widehat{\triangleright}\colon \triangleright\mathcal{U} \to \mathcal{U}$ satisfying

$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}(A))) \equiv \triangleright \mathsf{El}(A). \tag{2.3}$$

Using these, the type $\mathsf{Str}^g_{\mathbb{N}}$ can be defined as $\mathsf{El}(\widehat{\mathsf{Str}^g_{\mathbb{N}}})$ where

$$\widehat{\mathsf{Str}^g_{\mathbb{N}}} = \mathsf{fix}(\lambda B\colon \triangleright\mathcal{U}. \widehat{\mathbb{N}} \mathbin{\widehat{\times}} \widehat{\triangleright} B)$$

Note that this satisfies the expected type equality because

$$\begin{aligned} \mathsf{El}(\widehat{\mathsf{Str}^g_{\mathbb{N}}}) &\equiv \mathsf{El}(\widehat{\mathbb{N}} \mathbin{\widehat{\times}} \widehat{\triangleright}(\mathsf{next}(\widehat{\mathsf{Str}^g_{\mathbb{N}}}))) \\ &\equiv \mathsf{El}(\widehat{\mathbb{N}}) \times \mathsf{El}(\widehat{\triangleright}(\mathsf{next}(\widehat{\mathsf{Str}^g_{\mathbb{N}}}))) \\ &\equiv \mathbb{N} \times \triangleright \mathsf{El}(\widehat{\mathsf{Str}^g_{\mathbb{N}}}) \end{aligned}$$

Likewise, guarded recursive (proof-relevant) predicates on a type A, i.e., terms of type $A \to \mathcal{U}$ can be defined using fix as we shall see an example of in Section 2.5.

Note that we just assume a single universe and that the above only allows us to solve type equations that can be expressed as endomorphisms on this universe.[1] All the type equations considered in this paper are on this form, but we shall not always prove this explicitly, and often work with types rather than codes, in order to keep the presentation simple.

---

[1] It is also sound to add guarded recursive types as primitives to the type theory without use of universes, see [Bir+12]

$$\Gamma, x:\sigma, \Delta \vdash x:\sigma \qquad \frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash (\lambda x:\sigma.M):\sigma \to \tau}$$

$$\frac{\Gamma \vdash M:\sigma \to \tau \quad \Gamma \vdash N:\sigma}{\Gamma \vdash MN:\tau} \qquad \frac{\Gamma \vdash M:\sigma \to \sigma}{\Gamma \vdash \mathsf{Y}_\sigma\ M:\sigma}$$

$$\Gamma \vdash \underline{n}:\mathbf{nat} \qquad \frac{\Gamma \vdash M:\mathbf{nat}}{\Gamma \vdash \mathsf{succ}\ M:\mathbf{nat}} \qquad \frac{\Gamma \vdash M:\mathbf{nat}}{\Gamma \vdash \mathsf{pred}\ M:\mathbf{nat}}$$

$$\frac{\Gamma \vdash L:\mathbf{nat} \quad \Gamma \vdash M:\sigma \quad \Gamma \vdash N:\sigma}{\Gamma \vdash \mathsf{ifz}\ L\ M\ N:\sigma}$$

FIGURE 2.0.1: PCF typing rules

2.2.1 The topos of trees model

The type theory gDTT can be modelled in the topos of trees [Bir+12], i.e., the category of presheaves over ω, the first infinite ordinal. Since this is a topos, it is a model of extensional type theory. A closed type is modelled as a family of sets $X(n)$ indexed by natural numbers together with restriction maps $r_n\colon X(n+1) \to X(n)$. We think of $X(n)$ as how the type looks if we have n computation steps to reason about it. Using the propositions-as-types interpretation, we say that X is *true at stage* n if $X(n)$ is inhabited. Note that if X is true at stage n, it is also true at stage k for all $k \leq n$. Thus, the intuition of this model is that a proposition is initially considered true and can only be falsified by further computation.

In the topos of trees model, the $\triangleright$ modality is interpreted as $\triangleright X(0) = 1$ and $\triangleright X(n+1) = X(n)$, i.e., from the logical point of view, the $\triangleright$ modality delays evaluation of a proposition by one time step. For example, if 0 is the constantly empty presheaf (corresponding to a false proposition), then $\triangleright^n 0$ is the proposition that appears true for the first n computation steps and is falsified after $n+1$ steps.

2.3 PCF

This section defines the syntax, typing judgements, and operational semantics of PCF. These should be read as judgements in guarded type theory, but as stated above we work *informally* in type theory, which here means that we ignore standard problems of representing syntax up to α-equality. Note that this is a perpendicular issue to the one we are trying to solve here.

Unlike the operational semantics to be defined below, the typing judgements of PCF are defined in an entirely standard way, see Figure 2.0.1. In the figure, v ranges over values of PCF, i.e., terms of the form $v = \underline{n}$, where n is a natural number or $v = \lambda x.M$. Note that we distinguish notationally between a natural number n and the corresponding PCF value $\underline{n}$. We denote by `Type`$_{\text{PCF}}$, `Term`$_{\text{PCF}}$and `Value`$_{\text{PCF}}$ the types of PCF types, *closed* terms, and *closed* values of PCF.

$$
\begin{aligned}
\Downarrow : \texttt{Term}_{\texttt{PCF}} \times \mathbb{N} \times (\texttt{Value}_{\texttt{PCF}} \to \mathbb{N} \to \mathcal{U}) \to \mathcal{U} \\
v \Downarrow^k Q &\overset{\text{def}}{=\!=} Q(v,k) \\
\mathsf{pred}\ M \Downarrow^k Q &\overset{\text{def}}{=\!=} M \Downarrow^k (\lambda x.\lambda l.\Sigma n \colon \mathbb{N}.x = \underline{n} \text{ and } Q(\underline{n-1}, l)) \\
\mathsf{succ}\ M \Downarrow^k Q &\overset{\text{def}}{=\!=} M \Downarrow^k (\lambda x.\lambda l.\Sigma n \colon \mathbb{N}.x = \underline{n} \text{ and } Q(\underline{n+1}, l)) \\
\mathsf{Y}_\sigma\ M \Downarrow^k Q &\overset{\text{def}}{=\!=} \Sigma k'.k = k' + 1.\triangleright(M(\mathsf{Y}_\sigma\ M) \Downarrow^{k'} Q) \\
MN \Downarrow^k Q &\overset{\text{def}}{=\!=} M \Downarrow^k Q' \\
&\quad \text{where } Q'(\lambda x.L, m) = L[N/x] \Downarrow^m Q \\
\mathsf{ifz}\ L\ M\ N \Downarrow^k Q &\overset{\text{def}}{=\!=} L \Downarrow^k Q' \\
&\quad \text{where } Q'(\underline{0}, m) = M \Downarrow^m Q \text{ and } Q'(\underline{n+1}, m) = N \Downarrow^m Q
\end{aligned}
$$

FIGURE 2.0.2: Step-indexed Big-Step Operational Semantics for PCF

2.3.1 Big-step semantics

The big-step operational semantics defined in Figure 2.0.2 is a relation between terms, numbers and predicates on values and numbers. The statement $M \Downarrow^k Q$ should be read as M evaluates in $l \leq k$ steps to a value v satisfying $Q(v, k-l)$. The relation is defined as an inductive type. Note that, in the case of $\mathsf{pred}\ M$, the notation for $\underline{n-1}$ means n must also be greater than zero. If $Q : \texttt{Value}_{\texttt{PCF}} \to \mathcal{U}$ we overload notation and write

$$M \Downarrow^k Q \overset{\text{def}}{=\!=} M \Downarrow^k (\lambda \langle v, l \rangle .l = 0 \text{ and } Q(v)) \tag{2.4}$$

to be read as “M evaluates in exactly k steps to a value satisfying Q”. We can overload even further as follows

$$
\begin{aligned}
M \Downarrow^k v &\overset{\text{def}}{=\!=} M \Downarrow^k (\lambda w.w = v) \\
M \Downarrow v &\overset{\text{def}}{=\!=} \Sigma k.M \Downarrow^k v
\end{aligned}
$$

As mentioned in the introduction, the formulation of the big-step operational semantics is quite delicate – the wrong definition will make the adequacy theorem false. First of all, the definition must ensure that the steps of PCF are synchronised with the steps on the meta level. This is the reason for the use of $\triangleright$ in the case of the fixed point combinator. Secondly, the use of predicates on values on the right hand side of $\Downarrow$ rather than simply values is necessary to ensure that the right hand side is not looked at before the term is fully evaluated. For example, a naive definition of the operational semantics using values on the right hand side and the rule

$$\mathsf{succ}\ M \Downarrow^k v \overset{\text{def}}{=\!=} \Sigma n \colon \mathbb{N}.(v = \underline{n} + 1) \text{ and } M \Downarrow^k \underline{n}$$

Would make $(\mathsf{succ}\ (\mathsf{Y}_{\textbf{nat}}\ (\lambda x \colon \textbf{nat}.x)) \Downarrow^{42} \underline{0})$ false, but to obtain computational adequacy, we need this statement to be true for the first 42 steps before being falsified. (For an explanation of this point, see Remark 2.30 below.) In general, for $Q : \texttt{Value}_{\texttt{PCF}} \to \mathcal{U}$, $M \Downarrow^k Q$ should be defined in such a way

$$\frac{}{(\lambda x:\sigma.M)(N) \to^0 M[N/x]}\ (\mathsf{SLam}) \qquad \frac{}{\mathsf{Y}_\sigma\ M \to^1 M(\mathsf{Y}_\sigma\ M)}\ (\mathsf{SFix})$$

$$\frac{}{\mathsf{pred}\ \underline{0} \to^0 \underline{0}}\ (\mathsf{SPredZ}) \qquad \frac{}{\mathsf{pred}\ \underline{n+1} \to^0 \underline{n}}\ (\mathsf{SPredN})$$

$$\frac{}{\mathsf{succ}\ \underline{n} \to^0 \underline{n+1}}\ (\mathsf{SSucc})$$

$$\frac{}{\mathsf{ifz}\ \underline{0}\ M\ N \to^0 M}\ (\mathsf{SIfZ}) \qquad \frac{}{\mathsf{ifz}\ (\underline{n+1})\ M\ N \to^0 N}\ (\mathsf{SIfN})$$

$$\frac{L \to^k L'}{E[L] \to^k E[L']}\ (\mathsf{SIfZ})$$

$$E \in \mathtt{Context}_{\mathtt{PCF}} ::= [\cdot] \mid EM \mid \mathsf{pred}\ E \mid \mathsf{succ}\ E \mid \mathsf{ifz}\ E\ M\ N$$

FIGURE 2.0.3: Step-Indexed Small Step semantics of PCF. In the rules, k can be 0 or 1.

that in the topos of trees model it is true at stage n (using vocabulary from Section 2.2.1) iff either

- $k < n$ and M evaluates in precisely k steps to a value satisfying Q, or
- $k \geq n$ and evaluation of M takes more than k steps.

In particular, if M diverges, then $M \Downarrow^k Q$ should be true at stages $n \leq k$ and false for $n > k$.

The use of predicates means that partial results of term evaluation are ignored, and comparison of the result to the right hand side of $\Downarrow$ is postponed until evaluation of the term is complete. Moreover, the use of predicates also for the counting the steps is crucial to prove the correspondence between the two operational semantics and in particular for Lemma 2.5. This is because all the information about computations must be synchronised with the abstract notion of time. For example, if $MN \Downarrow^k Q$ we cannot decide a priori how many steps M will take to reduce to a value. We have to defer this information by waiting for the computation to run its course. When the terms reaches the value we pass the rest of the "fuel" to the predicate. So, a value reduces to a value along with possibly some additional steps to compute.

2.3.2 Small-step semantics

Figure 2.0.3 defines the small-step operational semantics. Just like the big step semantics, the small step semantics counts unfoldings of fixed points. The small steps semantics will be proved equivalent to the big-step semantics, but is introduced because it is more suitable for the proofs of soundness and computational adequacy.

Note the following easy lemma.

Lemma 2.1. *The small-step semantics is deterministic: if $M \to^k N$ and $M \to^{k'} N'$, then $k = k'$ and $N = N'$.*

The transitive closure of the small step semantics is defined using $\triangleright$ to ensure that the steps of PCF are synchronised with the steps of the meta language.

Definition 2.2. *Denote by* $\rightarrow^0_*$ *the reflexive, transitive closure of* $\rightarrow^0$*. The closure of the small step semantics, written* $M \Rightarrow^k Q$ *is a relation between closed terms, natural numbers, and predicates on closed terms, defined by induction on* k *as*

$$\begin{aligned} M \Rightarrow^0 Q &\overset{\text{def}}{=\!=} \Sigma N\colon \mathtt{Term}_{\mathtt{PCF}}.M \rightarrow^0_* N \textit{ and } Q(N) \\ M \Rightarrow^{k+1} Q &\overset{\text{def}}{=\!=} \Sigma M', M''\colon \mathtt{Term}_{\mathtt{PCF}}.M \rightarrow^0_* M' \textit{ and } M' \rightarrow^1 M'' \\ &\qquad \textit{and } \triangleright(M'' \Rightarrow^k Q) \end{aligned} \tag{2.5}$$

Similarly to the case of the big-step semantics we define $M \Rightarrow^k v \overset{\text{def}}{=\!=} M \Rightarrow^k \lambda N.v = N$

Operational Correspondence We now state the equivalence of the two operational semantics given above. However, since the big-step operational semantics in (2.4) is stated on values and the transitive closure on the small-step semantics in (2.2) is stated using terms, we first introduce the notation: for a predicate $Q : \mathtt{Value}_{\mathtt{PCF}} \rightarrow \mathcal{U}$,

$$Q_T(N) \overset{\text{def}}{=\!=} \Sigma v.N = v \text{ and } Q(v) \tag{2.6}$$

Note that $Q_T : \mathtt{Term}_{\mathtt{PCF}} \rightarrow \mathcal{U}$ whenever $Q : \mathtt{Value}_{\mathtt{PCF}} \rightarrow \mathcal{U}$. Now we can state the correspondence.

Lemma 2.3. *If* $M : \mathtt{Term}_{\mathtt{PCF}}$ *and* $Q : \mathtt{Value}_{\mathtt{PCF}} \rightarrow \mathcal{U}$*, then* $M \Downarrow^k Q$ *iff* $M \Rightarrow^k Q_T$

The following is the standard statement for operational correspondence and follows directly from Lemma 2.3.

Corollary 2.4. $M \Downarrow^k v \Leftrightarrow M \Rightarrow^k v$

We will now prove the correspondence between the big-step and the small step operational semantics. First we need the following lemma.

Lemma 2.5. *Let* M, N *be closed terms of type* τ*, and let* $Q : \mathtt{Value}_{\mathtt{PCF}} \times \mathbb{N} \rightarrow \mathcal{U}$*.*

1. *If* $M \rightarrow^0 N$ *and* $N \Downarrow^k Q$ *then* $M \Downarrow^k Q$
2. *If* $M \rightarrow^1 N$ *and* $\triangleright(N \Downarrow^k Q)$ *then* $M \Downarrow^{k+1} Q$

Proof. (Proof sketch)

1. By induction on $M \rightarrow^0 N$. We consider the case $\mathsf{ifz}\ L\ M\ N \rightarrow^0 \mathsf{ifz}\ L'\ M\ N$. Assume $\mathsf{ifz}\ L'\ M\ N \Downarrow^k Q$. By definition $L' \Downarrow^k Q'$. By induction hypothesis $L \Downarrow^k Q'$ and by definition $\mathsf{ifz}\ L\ M\ N \Downarrow^k Q$. All the other cases are similar.
2. By induction on $M \rightarrow^1 N$. We sketch a few cases. For the case $\mathsf{Y}_\sigma\ M \rightarrow^1 M(\mathsf{Y}_\sigma\ M)$, assume $\triangleright(M(\mathsf{Y}_\sigma\ M) \Downarrow^k Q)$. Then by definition $\mathsf{Y}_\sigma\ M \Downarrow^{k+1} Q$.

   We consider now some inductive cases. For the case $M_1 N \rightarrow^1 M_2 N$, assume $\triangleright(M_2 N \Downarrow^k Q)$. By definition this is equivalent to $\triangleright(M_2 \Downarrow^k Q')$ where $Q'(\lambda x.L, l) = L[N/x] \Downarrow^l Q$. By induction hypothesis we get $M_1 \Downarrow^{k+1} Q'$ which is by definition what we wanted. For the case $\mathsf{pred}\ M \to^1 \mathsf{pred}\ M'$. Assume $\triangleright(\mathsf{pred}\ M' \Downarrow^k Q)$. By definition $\triangleright(M' \Downarrow^k \lambda\langle x,l\rangle.Q(x-1,l))$ and by induction hypothesis $M \Downarrow^{k+1} \lambda\langle x,l\rangle.Q(x-1,l)$. By definition $\mathsf{pred}\ M \Downarrow^{k+1} Q$.

$\square$

The following lemma follows almost straightforwardly from the previous one.

Lemma 2.6. *For all* $M N : \mathtt{Term}_{\mathtt{PCF}}$*, if* $M \to^0_* N$ *and* $N \Downarrow^k v$ *then* $M \Downarrow^k v$

Proof. By induction on $M \to^0_* N$. The base case $M = N$ is straightforward. Now the inductive case $M \to^0 N'$ and $N' \to^0_* N$. By induction hypothesis $N' \Downarrow^k v$. By Lemma 2.5 we conclude. $\square$

However, we cannot prove directly Lemma 2.3 which states the correspondence between the big-step and the small-step operational semantics. This is because the right hand side of the big-step semantics in Figure 2.4 states convergence at most with some k steps whereas the small-step in Definition 2.2 – when stated with values on right hand side – states convergence in exactly some number of steps. For this reason we need an intermediate formulation of the transitive closure which we give as follows:

Definition 2.7. *Let* Q *be a predicate of type* $\mathtt{Term}_{\mathtt{PCF}} \times \mathbb{N} \to \mathcal{U}$ *and define* $M \Rrightarrow^k Q$ *as an inductive dependent type as follows*

$$\frac{\Sigma N : \mathtt{Term}_{\mathtt{PCF}}.M \to^0_* N \text{ and } Q(N,k)}{M \Rrightarrow^k Q}$$

$$\frac{\Sigma M' M'' : \mathtt{Term}_{\mathtt{PCF}}.M \to^0_* M' \text{ and } M' \to^1 M'' \text{ and } \triangleright(M'' \Rrightarrow^k Q)}{M \Rrightarrow^{k+1} Q}$$

The following proposition follows straightforwardly from the definition.

Proposition 2.8. *For all* $M, N : \mathtt{Term}_{\mathtt{PCF}}$ *and* $Q : \mathtt{Term}_{\mathtt{PCF}} \times \mathbb{N} \to \mathcal{U}$*, if* $M \to^0_* N$ *and* $N \Rrightarrow^k Q$ *then* $M \Rrightarrow^k Q$

The small-step semantics are compositional in the following sense.

Lemma 2.9. *For all* $M, N : \mathtt{Term}_{\mathtt{PCF}}$*, if* $M \Rrightarrow^k Q'$ *with* $Q'(L,n) = L \Rrightarrow^n Q$ *then* $M \Rrightarrow^k Q$

Proof. By induction on $M \Rrightarrow^k Q'$. In the first case we have that $M \to^0_* N$ and $Q'(N,k)$, i.e. $M \Rrightarrow^k Q$, so by Proposition 2.8 we get $M \Rrightarrow^k Q$. Now the second case. By assumption we get $M \to^0_* M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rrightarrow^{k-1} Q')$. By induction hypothesis we get $\triangleright(M'' \Rrightarrow^{k-1} Q)$ which together with $M \to^0_* M'$ and $M' \to^1 M''$ give by definition $M \Rrightarrow^k Q$. $\square$

The small-step semantics as in Definition 2.7 behaves well w.r.t. the contexts. To make this statement precise we define, for some context E as in Figure 2.0.3 and for some predicate Q on terms, a predicate Q_E as follows:

$$Q_E(T,k) \stackrel{\text{def}}{=\!=} \Sigma M.T = E[M] \text{ and } Q(M,k)$$

Intuitively, Q_E is true for terms of the form $E[M]$ for some M that satisfies Q. We use $Q_E(E[M], k)$ to explicitly indicate that the converging terms is of the form $E[M]$ and that M is the term satisfying Q. We prove now that $\Rrightarrow$ is closed under context application.

Lemma 2.10. *For all* M : $\texttt{Term}_{\texttt{PCF}}$*, if* $M \Rrightarrow^m Q$ *then* $E[M] \Rrightarrow^m Q_E$ *with* $Q_E(E[M'], m) = Q(M', m)$*.*

Proof. The proof is by induction on $M \Rrightarrow^m Q$. For the first case there exists M' such that $M \to^0_* M'$ and that satisfies the predicate $Q(M', m)$. By easy induction on the relation $\to^0_*$ we get $E[M] \to^0_* E[M']$. Now, $E[M']$ is indeed of the form required by Q_E and M' satisfies $Q(M', m)$, hence $E[M] \Rightarrow^k 0Q_E$.

As for the second case there exists m' s.t. $m = m' + 1$ and there exists an M' and M'' such that $M \to^0_* M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rrightarrow^{m'} Q)$. By assumption we get $E[M] \to^0_* E[M']$ and $E[M'] \to^1 E[M'']$. By induction hypothesis we get $\triangleright(E[M''] \Rrightarrow^{m'} Q_E)$ and by definition $E[M] \Rrightarrow^{m+1} Q_E$, thus concluding the case. □

We prove that the small-step operational semantics is sound w.r.t the big-step.

Lemma 2.11. *Let* M *be a closed term and* $Q : \texttt{Value}_{\texttt{PCF}} \times \mathbb{N} \to \mathcal{U}$ *a relation on values. If* $M \Rrightarrow^k (\lambda N.\lambda z.(N \Ddownarrow^z Q))$ *then* $M \Ddownarrow^k Q$

Proof. The proof is by induction on $M \Rrightarrow^k (\lambda N.\lambda z.(N \Ddownarrow^z Q))$.

First case is straightforward from Lemma 2.6.

Now we prove the second case. We have that $M \to^0_* M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rrightarrow^{k'} \lambda \langle N, z \rangle .(N \Ddownarrow^z Q))$. By induction hypothesis we know that $\triangleright(M'' \Ddownarrow^{k'} Q)$ and by Lemma 2.5 and Lemma 2.6 we obtain $M \Ddownarrow^k Q$. □

In the following lemma we are going to prove that the big-step operational semantics correspond to the small-step. To this end, we overload the lifting of the predicates (2.6) as follows: for a predicate $Q : \texttt{Value}_{\texttt{PCF}} \times \mathbb{N} \to \mathcal{U}$,

$$Q_T \stackrel{\text{def}}{=\!=} \lambda \langle N, k \rangle .\Sigma v.N = v \text{ and } Q(v, k)$$

Also we are going to make use of the fact that $M \Ddownarrow^k -$ is covariant in the following sense:

Proposition 2.12. *Let* Q *and* R *two predicates on values. If* Q *implies* R *then* $M \Ddownarrow^k Q$ *implies* $M \Ddownarrow^k R$*.*

Now we can prove that the big-step operational semantics correspond to the intermediate definition of small-step semantics.

Lemma 2.13. *If* M : $\texttt{Term}_{\texttt{PCF}}$ *and* Q : $\texttt{Value}_{\texttt{PCF}} \times \mathbb{N} \to \mathcal{U}$*, then* $M \Ddownarrow^k Q$ *iff* $M \Rrightarrow^k Q_T$

Proof. We first prove that if $M \Ddownarrow^k Q$ then $M \Rrightarrow^k Q_T$ by induction on $M \Ddownarrow^k Q$. Here we sketch the main cases. For the case $v \Ddownarrow^k Q$ by definition $Q(v, k)$ is inhabited. This together with the fact that $v \to^0_* v$ by reflexivity give us $v \Rrightarrow^k \lambda N.\lambda k.\Sigma v.N = v$ and $Q(v, k)$.

For the case of $MN \Ddownarrow^k Q$ by definition $M \Ddownarrow^k Q^1$, where $Q^1(\lambda x.L, m) = L[N/x] \Ddownarrow^m Q$. By induction hypothesis on $L[N/x] \Ddownarrow^m Q$ we get that Q^1 implies

$$Q^2(\lambda x.L, m) = L[N/x] \Rrightarrow^m Q_T$$

thus $M \Downarrow^k Q^2$. Since $(\lambda x.L)N \to^0_* L[N/x]$, by applying Proposition 2.8 we get that Q^2 implies

$$Q^3(\lambda x.L, m) = (\lambda x.L)N \Rrightarrow^m Q_T$$

thus $M \Downarrow^k Q^3$. By applying the induction hypothesis on M we get $M \Rrightarrow^k Q^3_T$ which is equivalent to $M \Rrightarrow^k Q^3$. By applying Lemma 2.10 with context $[-]N$ we get $MN \Rrightarrow^k Q^4$ where Q^4 is defined as

$$Q^4((\lambda x.L)N, m) = Q^3(\lambda x.L, m)$$

which is equal to

$$Q^4((\lambda x.L)N, m) = (\lambda x.L)N \Rrightarrow^m Q_T$$

Directly from Lemma 2.9 we get $MN \Rrightarrow^k Q_T$.

For the case when $\mathsf{ifz}\ L\ M\ N \Downarrow^k Q$ by definition we get $L \Downarrow^k Q^1$ where

$$Q^1(v, m) = (v = 0 \text{ and } M \Downarrow^m Q \text{ or } (v = n+1 \text{ and } N \Downarrow^m Q))$$

By induction hypothesis on M and N together with Proposition 2.12 we get $L \Downarrow^k Q^2$ where

$$Q^2(v, m) = (v = 0 \text{ and } M \Rrightarrow^m Q_T \text{ or } (v = n+1 \text{ and } N \Rrightarrow^m Q_T$$

By Lemma 2.10 and Proposition 2.12 $L \Downarrow^k Q^3$

$$Q^3(v, m) = \mathsf{ifz}\ v\ M\ N \Rightarrow^m Q_T$$

By induction hypothesis on L and Proposition 2.12 $L \Rightarrow^k Q^3_T$ and again by Lemma 2.10 $\mathsf{ifz}\ L\ M\ N \Rightarrow^k Q$.

The most interesting case is the fixed point case. Assume $\mathsf{Y}_\sigma\ M \Downarrow^{k+1} Q$. By definition $\triangleright(M\,\mathsf{Y}_\sigma\ M \Downarrow^k Q)$. By induction hypothesis $\triangleright(M\,\mathsf{Y}_\sigma\ M \Rrightarrow^k Q_T)$. As $\mathsf{Y}_\sigma\ M \to^1 M\,\mathsf{Y}_\sigma\ M$ by Definition 2.7 $\mathsf{Y}_\sigma\ M \Rrightarrow^{k+1} Q_T$ We prove now that if $M \Rrightarrow^k Q_T$ then $M \Downarrow^k Q$. Assume $M \Rrightarrow^k Q_T$. Since Q_T implies Q' where

$$Q'(N, k) = N \Downarrow^k Q$$

We can apply Lemma 2.11 thus getting $M \Downarrow^k Q$. □

We now prove the two definitions for the small-step semantics coincide. To do this we have to lift the predicate on values and steps as follows: for a predicate $Q : \mathtt{Term}_{\mathtt{PCF}} \to \mathcal{U}$ define

$$Q_0(N, k) \overset{\text{def}}{=\!=} k = 0 \text{ and } Q(N)$$

Intuitively, Q_0 considers only reductions when the remaining number of steps is zero.

Lemma 2.14. *For all PCF terms M and $Q : \mathtt{Term}_{\mathtt{PCF}} \to \mathcal{U}$, $M \Rrightarrow^k Q_0$ iff $M \Rightarrow^k Q$.*

Proof. We first prove the left-to-right direction by induction on $M \Rrightarrow^k Q_0$. The base case is when $M \to^0_* N$ and $Q_0(N, k)$. The latter implies $k = 0$ and

$Q(N)$, so we have to prove $M \Rightarrow^0 Q$ which is straightforward. The inductive case follows by definition.

We now prove the right-to-left direction assuming that $M \Rightarrow^k Q$ and proceeding by induction on k. The base case is $k = 0$, therefore $M \to^0_* N$ and $Q(N)$. by assumption $Q_0(N, k)$ is true. Hence $M \Rrightarrow^k Q_0$. The inductive case is $k = k' + 1$. By definition $M \to^0_* M'$ and $M' \to^1 M''$ and $\triangleright(M'' \Rightarrow^{k'} Q)$. By induction $\triangleright(M'' \Rrightarrow^{k'} Q_0)$ and by definition $M \Rrightarrow^k Q_0$. □

Now we can prove Lemma 2.3.

Proof of Lemma 2.3.

- $M \Downarrow^k Q_0$ iff $M \Rrightarrow^k Q_0$ is a particular instance of Lemma 2.13
- $M \Rrightarrow^k Q_0$ iff $M \Rightarrow^k Q$ is by Lemma 2.14

□

2.4 Denotational semantics

We now define the denotational semantics of PCF. For this, we use the guarded recursive *lifting monad* on types, defined as the guarded recursive type[2]

$$LA \overset{\text{def}}{=\!=} \mathsf{fix}\, X.(A + \triangleright X).$$

Let $i \colon A + \triangleright LA \cong LA$ be the isomorphism, let $\theta : \triangleright LA \to LA$ be the right inclusion composed with i and let $\eta \colon A \to LA$ (the unit of the monad) denote the left inclusion composed with i. Note that any element of LA is either of the form $\eta(a)$ or $\theta(r)$. We can describe the universal property of LA as follows. Define a $\triangleright$-algebra to be a type B together with a map $\theta_B : \triangleright B \to B$. The lifting LA as defined above is the *free* $\triangleright$-algebra on A. Given $f : A \to B$ with B a $\triangleright$-algebra, the unique extension of f to a homomorphism of $\triangleright$-algebras $\hat{f} \colon LA \to B$ is defined as

$$\hat{f}(\eta(a)) \overset{\text{def}}{=\!=} f(a)$$
$$\hat{f}(\theta(r)) \overset{\text{def}}{=\!=} \theta_B(\mathsf{next}(\hat{f}) \circledast r)$$

which can be formally expressed as a fixed point of a term of type $\triangleright(LA \to B) \to LA \to B$.

The intuition the reader should have for L is that LA is the type of computations possibly returning an element of A, recording the number of steps used in the computation. The unit η gives an inclusion of values into computations, the composite $\delta = \theta \circ \mathsf{next} \colon LA \to LA$ is an operation that adds one time step to a computation, and the bottom element $\bot = \mathsf{fix}(\theta)$ is the diverging computation. In fact, any $\triangleright$-algebra has a bottom element and an operation δ as defined above, and homorphisms preserve this structure. The lifting L extends to a functor. For a map $f : A \to B$ the action on morphisms can be defined using the unique extension as $L(f) \overset{\text{def}}{=\!=} \widehat{\eta \circ f}$.

---

[2] Since guarded recursive types are encoded using universes, L is strictly an operation on $\mathcal{U}$. We will only apply L to types that have codes in $\mathcal{U}$.

$$\begin{aligned}
\llbracket x_1 : \sigma_1, \cdots, x_k : \sigma_k \vdash x_i \rrbracket (\gamma) &= \pi_i \gamma \\
\llbracket \Gamma \vdash \underline{n} : \mathbf{nat} \rrbracket (\gamma) &= \eta(n) \\
\llbracket \Gamma \vdash \mathsf{Y}_\sigma\ M \rrbracket (\gamma) &= (\mathsf{fix}_{\llbracket \sigma \rrbracket})(\lambda x : \triangleright \llbracket \sigma \rrbracket . \theta_\sigma(\mathsf{next}(\llbracket M \rrbracket (\gamma))) \circledast x)) \\
\llbracket \Gamma \vdash \lambda x . M \rrbracket (\gamma) &= \lambda x . \llbracket M \rrbracket (\gamma, x) \\
\llbracket \Gamma \vdash M N \rrbracket (\gamma) &= \llbracket M \rrbracket (\gamma) \llbracket N \rrbracket (\gamma) \\
\llbracket \Gamma \vdash \mathsf{succ}\ M \rrbracket (\gamma) &= L(\lambda x . x + 1)(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \mathsf{pred}\ M \rrbracket (\gamma) &= L(\lambda x . x - 1)(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \mathsf{ifz}\ L\ M\ N \rrbracket (\gamma) &= (\widehat{\mathsf{ifz}}(\llbracket M \rrbracket (\gamma), \llbracket N \rrbracket (\gamma)))(\llbracket L \rrbracket (\gamma))
\end{aligned}$$

FIGURE 2.14.1: Interpretation of terms

2.4.1 Interpretation

The interpretation function $\llbracket \cdot \rrbracket : \mathtt{Type}_{\mathtt{PCF}} \to \mathcal{U}$ is defined by induction.

$$\llbracket \mathbf{nat} \rrbracket \overset{\text{def}}{=\!=} L\mathbb{N}$$

$$\llbracket \tau \to \sigma \rrbracket \overset{\text{def}}{=\!=} \llbracket \tau \rrbracket \to \llbracket \sigma \rrbracket$$

The denotation of every type is a $\triangleright$-algebra: the map $\theta_\sigma \colon \triangleright \llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket$ is defined by induction on σ by

$$\theta_{\sigma \to \tau} = \lambda f \colon \triangleright(\llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket) . \lambda x \colon \llbracket \sigma \rrbracket . \theta_\tau(f \circledast \mathsf{next}(x))$$

Typing judgements $\Gamma \vdash M : \sigma$ are interpreted as usual as functions from $\llbracket \Gamma \rrbracket$ to $\llbracket \sigma \rrbracket$, where the interpretation of contexts is defined as

$$\llbracket x_1 : \sigma_1, \cdots, x_k : \sigma_k \rrbracket \overset{\text{def}}{=\!=} \llbracket \sigma_1 \rrbracket \times \cdots \times \llbracket \sigma_n \rrbracket$$

Figure 2.14.1 defines the interpretation of judgements. Below we often write $\llbracket M \rrbracket$ rather than $\llbracket \Gamma \vdash M \colon \sigma \rrbracket$. Natural numbers in PCF are computations that produce a value in zero steps, so we interpret them by using $\langle\rangle$. In the case of Y_σ we have by induction a map $\llbracket M \rrbracket (\gamma)$ of type $\llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket$. Morally, $\llbracket \Gamma \vdash \mathsf{Y}_\sigma\ M \rrbracket (\gamma)$ should be the fixed point of $\llbracket M \rrbracket (\gamma)$ composed with δ, ensuring that each unfolding of the fixed point is recorded as a step in the model, but to get the types correct, we have to apply the functorial action of $\triangleright$ to $\llbracket M \rrbracket (\gamma)$ and compose with θ instead of δ. The intuition given above is captured in the following lemma.

Lemma 2.15. *Let* $\Gamma \vdash M : \sigma \to \sigma$ *then* $\llbracket \mathsf{Y}_\sigma\ M \rrbracket = \delta \circ \llbracket M(\mathsf{Y}_\sigma\ M) \rrbracket$

Proof. Let $\gamma : \llbracket \Gamma \rrbracket$. By definition $\llbracket \mathsf{Y}_\sigma\ M \rrbracket (\gamma)$ is equal to

$$(\mathsf{fix}_{\llbracket \sigma \rrbracket})(\lambda x : \triangleright \llbracket \sigma \rrbracket . \theta_\sigma(\mathsf{next}(\llbracket M \rrbracket (\gamma)) \circledast x))$$

We unfold the fixed point thus getting

$$(\lambda x . \theta_\sigma(\mathsf{next}(\llbracket M \rrbracket (\gamma)) \circledast x))(\mathsf{next}(\mathsf{fix}(F)))$$

By function application we get $\theta_\sigma(\mathsf{next}(\llbracket M \rrbracket(\gamma)) \circledast \mathsf{next}(\mathsf{fix}(F)))$. Since next is a natural we get $\theta_\sigma(\mathsf{next}(\llbracket M \rrbracket(\gamma)(\mathsf{fix}(F))))$ which is by definition equal to $\delta_\sigma(\llbracket M\, \mathsf{Y}_\sigma\ M \rrbracket(\gamma))$. ☐

We now explain the interpretation of ifz $L\ M\ N$. Define first a semantic $\mathsf{ifz}\colon \llbracket\sigma\rrbracket \to \llbracket\sigma\rrbracket \to \mathbb{N} \to \llbracket\sigma\rrbracket$ operation by

$$\mathsf{ifz}\, x\ y\ 0 \stackrel{\text{def}}{=\!=} x \qquad\qquad \mathsf{ifz}\, x\ y\ (n+1) \stackrel{\text{def}}{=\!=} y$$

The operation $\widehat{\mathsf{ifz}}\colon \llbracket\sigma\rrbracket \to \llbracket\sigma\rrbracket \to \llbracket\mathbf{nat}\rrbracket \to \llbracket\sigma\rrbracket$ is defined by $\widehat{\mathsf{ifz}}\ x\ y$ being the extension of ifz $x\ y$ to a homomorphism of $\triangleright$-algebras. As a direct consequence of this definition we get

Lemma 2.16. *1.*

$$\llbracket\, \lambda x\colon \mathbf{nat}.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\theta(r)) = \theta(\mathsf{next}(\llbracket\, \lambda x\colon \mathbf{nat}.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma)) \circledast r) \tag{2.7}$$

2. *If* $\llbracket L \rrbracket(\gamma) = \delta(\llbracket L' \rrbracket(\gamma))$, *then*

$$\llbracket \mathsf{ifz}\ L\ M\ N \rrbracket(\gamma) = \delta \left\llbracket \mathsf{ifz}\ L'\ M\ N \right\rrbracket(\gamma) \tag{2.8}$$

Proof. First we prove (2.7). By definition $\llbracket \lambda x.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma)(\theta(r))$ is equal to

$$(\lambda x.\, \llbracket \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma, x))(\theta(r))$$

The interpretation of ifz is defined as a fixed point, namely,

$$(\lambda x.\, \mathsf{fix}(\widehat{\mathsf{ifz}}(\llbracket M \rrbracket(\gamma, x), \llbracket N \rrbracket(\gamma, x)))(x))(\theta(r))$$

By simplified we obtain $\mathsf{fix}(\widehat{\mathsf{ifz}}(\llbracket M \rrbracket(\gamma), \llbracket N \rrbracket(\gamma)))(\theta(r))$ By unfolding the guarded fixed-point (rule (2.2) we obtain

$$(\widehat{\mathsf{ifz}}(\llbracket M \rrbracket(\gamma), \llbracket N \rrbracket(\gamma))(\mathsf{next}(\mathsf{fix}(\mathsf{ifz}(\llbracket M \rrbracket(\gamma), \llbracket N \rrbracket(\gamma)))))(\theta(r)))$$

By definition $\widehat{\mathsf{ifz}}$ applied to $\theta(r)$ is

$$\theta_\sigma((\mathsf{next}(\mathsf{fix}(\mathsf{ifz}(\llbracket M \rrbracket(\gamma), \llbracket N \rrbracket(\gamma))))) \circledast r \circledast \mathsf{next}(\llbracket M \rrbracket(\vec{\alpha})) \circledast \mathsf{next}(\llbracket N \rrbracket(\vec{\alpha})))$$

By (2.1) we group up the nexts thus getting

$$\theta_\sigma((\mathsf{next}\, \llbracket \lambda x.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma)) \circledast r)$$

We prove now (2.8). For a variable $x : \llbracket\mathbf{nat}\rrbracket$, by definition of δ we know that

$$\llbracket \lambda x.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma)(\delta_{\mathbf{nat}}(x)) = \llbracket \lambda x.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma)(\theta_{\mathbf{nat}}(\mathsf{next}(x)))$$

By (2.7) we can pull out the θ and group up the nexts by rule (2.1)

$$(\theta_\sigma(\mathsf{next}(\llbracket \lambda x.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma))(x)))$$

which is by definition

$$(\delta_\sigma(\llbracket \lambda x.\, \mathsf{ifz}\ x\ M\ N \rrbracket(\gamma))(x))$$

□

2.4.2 Soundness

The soundness theorem states that if a program M evaluates to a value v in k steps then the interpretation of M is equal to the interpretation of v delayed k times by the semantic delay operation δ. Thus the soundness theorem captures not just extensional but also intensional behaviour of terms.

First we state the Substitution Lemma needed for the cases of the function application.

Proposition 2.17 (Substitution Lemma). *Let* $\Gamma \equiv x_1 : \sigma_1, \cdots, x_k : \sigma_k$ *be a context for* $\Gamma \vdash M : \tau$*, for all* $\Delta \vdash N_i : \sigma_i$ *for* $i = 1, \cdots, k$*, for all* $\vec{d} \in \llbracket \Delta \rrbracket$*,*
$$\left\llbracket \Delta \vdash M[\vec{N}/x] : \tau \right\rrbracket (\vec{d}) = \llbracket \Gamma \vdash M : \tau \rrbracket \left(\llbracket \Delta \vdash N_1 : \sigma_1 \rrbracket (\vec{d}), \cdots, \llbracket \Delta \vdash N_k : \sigma_k \rrbracket (\vec{d})\right)$$

The soundness theorem is proved using the small-step semantics. We first need a lemma for the single step reduction.

Lemma 2.18. *Let* M *be a closed term of type* τ*. If* $M \to^k N$ *then* $\llbracket M \rrbracket (*) = \delta^k \llbracket N \rrbracket (*)$

Proof. The proof goes by induction on $M \to^k N$.

We first prove the bases cases. The first case is the function application, namely $(\lambda x.M)N \to^0 M[N/x]$. By the Substitution Lemma 2.17 we know that $\llbracket (\lambda x.M)N \rrbracket (*) = \llbracket M[N/x] \rrbracket (*)$ holds concluding the case. The case for the fixed-point combinator, namely $\mathsf{Y}_\sigma\ M \to^1 M(\mathsf{Y}_\sigma\ M)$, follows from Lemma 2.15. Now the zero case for the conditional statement: $\mathsf{ifz}\ 0\ M\ N \to^0 M$. By definition

$$\llbracket \mathsf{ifz}\ \underline{0}\ M\ N \rrbracket (*) = (\widehat{\mathsf{ifz}}(\llbracket M \rrbracket (*), \llbracket N \rrbracket (*)))(\llbracket \underline{0} \rrbracket (*))$$

Since $\llbracket \underline{0} \rrbracket (*) = \eta(0)$, the above is equal, by definition of $\widehat{\mathsf{ifz}}$, to $\llbracket M \rrbracket (*)$. The case for $\mathsf{ifz}\ \underline{n+1}\ M\ N \to^0 N$ is similar. Now the case for $\mathsf{pred}\ 0 \to^0 0$. By definition $\llbracket \mathsf{pred}\ 0 \rrbracket (*) = L(\lambda x.x - 1)(\llbracket 0 \rrbracket (*))$. Since $(\llbracket 0 \rrbracket (*)) = \eta(0)$, we get that $L(\lambda x.x - 1)(\llbracket 0 \rrbracket (*)) = \llbracket 0 \rrbracket (*)$. The case $\mathsf{pred}\ n + 1 \to^0 n$ is similar to previous case.

We prove now the inductive cases. For the function application $MN \to^k M'N$ by definition we get $\llbracket MN \rrbracket (*) = \llbracket M \rrbracket (*) \llbracket N \rrbracket (*)$ By induction hypothesis we get $(\delta_{\sigma \to \tau} \llbracket M' \rrbracket (*)) \llbracket N \rrbracket (*)$ which by definition of $\delta_{\sigma \to \sigma}$ is $\delta_\sigma(\llbracket M' \rrbracket (*) \llbracket N \rrbracket (*))$. Now the case for $\mathsf{pred}\ M \to^k \mathsf{pred}\ M'$. By definition $\llbracket \mathsf{pred}\ M \rrbracket (*) = L(\lambda x.x - 1)(\llbracket M \rrbracket (*))$. By induction hypothesis we know that $L(\lambda x.x - 1)(\delta^k \llbracket M' \rrbracket (*))$. By definition of the functorial action of L, we get that $\delta^k L(\lambda x.x - 1)(\llbracket M' \rrbracket (*))$ which is equal to $\delta^k \llbracket \mathsf{pred}\ M' \rrbracket (*)$. The proofs for $\mathsf{succ}\ M$ are similar to $\mathsf{pred}\ M$.

Now the case for the conditional statement $\mathsf{ifz}\ L\ M\ N \to^k \mathsf{ifz}\ L'\ M\ N$. By definition $\llbracket \mathsf{ifz}\ L\ M\ N \rrbracket (*) = (\widehat{\mathsf{ifz}}(\llbracket M \rrbracket (*), \llbracket N \rrbracket (*)))(\llbracket L \rrbracket (*))$ By induction hypothesis we get

$$(\widehat{\mathsf{ifz}}(\llbracket M \rrbracket (*), \llbracket N \rrbracket (*)))(\delta^k(\llbracket L' \rrbracket (*)))$$

We can pull out one δ by Lemma 2.16 thus getting

$$\delta^k((\widehat{\mathsf{ifz}}(\llbracket M \rrbracket (*), \llbracket N \rrbracket (*)))(\llbracket L' \rrbracket (*)))$$

which is equal by definition to $\delta^k \llbracket \mathsf{ifz}\ L'\ M\ N \rrbracket (*)$ □

Lemma 2.19. *Let M be a closed term of type τ. If $M \to^0_* N$, for some term N, then $\llbracket M \rrbracket (*) = \llbracket N \rrbracket (*)$*

Proof. The proof is straightforward. □

Now we extend the previous results to $\Rightarrow^k$.

Lemma 2.20. *Let M be a closed term of type τ, if $M \Rightarrow^k N$ then $\llbracket M \rrbracket (*) = \delta^k \llbracket N \rrbracket (*)$*

Proof. By induction on k. The case $k = 0$ follows from Lemma 2.18. Assume $k = k' + 1$. By definition we have $M \to^0_* M'$ and $M' \to^1 M''$ and $\triangleright(M'' \Rightarrow^{k'} N)$. By repeated application of Lemma 2.18 we get $\llbracket M \rrbracket (*) = \llbracket M' \rrbracket (*)$ and $\llbracket M' \rrbracket (*) = \delta(\llbracket M'' \rrbracket (*))$.

By induction hypothesis we get $\triangleright(\llbracket M'' \rrbracket (*) = \delta^{k'} \llbracket N \rrbracket (*))$. By gDTT rule $\mathsf{TY} - \mathsf{COM}_\triangleright$ this implies $\mathsf{next}(\llbracket M'' \rrbracket (*)) = \mathsf{next}(\delta^{k'} \llbracket N \rrbracket (*))$ and since $\delta = \theta \circ \mathsf{next}$, this implies $\delta \llbracket M'' \rrbracket (*) = \delta^k \llbracket N \rrbracket (*)$. By putting together the equations we get finally $\llbracket M \rrbracket (*) = \delta^k \llbracket N \rrbracket (*)$. □

The Soundness theorem follows from the fact that the small-step semantics is equivalent to the big step, which is Corollary 2.4.

Theorem 2.21 (Soundness)**.** *Let M be a closed term of type τ, if $M \Downarrow^k v$ then $\llbracket M \rrbracket (*) = \delta^k \llbracket v \rrbracket (*)$*

2.5 Computational Adequacy

In this section we prove that the denotational semantics is computationally adequate with respect to the operational semantics. At a high level, we proceed in the standard way, by constructing a logical relation $\mathcal{R}_\sigma$ between denotations $\llbracket \sigma \rrbracket$ and terms $\mathtt{Term}_{\mathtt{PCF}}$ and then proving that open terms and their denotation respect this relation (Lemma 2.28 below). We define our logical relation in guarded dependent type theory, so formally, it will be a map into the universe $\mathcal{U}$ of types. Thus we work with a proof-relevant logical relation, similar to what was recently done in work of Benton et. al. [BHN14].

To formulate the definition of the logical relations and also to carry out the proof of the fundamental theorem of logical relations, we need some more sophisticated features of gDTT, which we now recall.

2.5.1 Guarded Dependent Type Theory

We recall some key features of gDTT; see [Biz+16] for more details.

As mentioned in Section 2.2, the later functor $\triangleright$ is an applicative functor. Guarded dependent type theory extends the later application $\circledast \colon \triangleright(A \to B) \to \triangleright A \to \triangleright B$ to the dependent case using a new notion of *delayed substitution*: if $\Gamma \vdash f : \triangleright\Pi(x : A).B$ and $\Gamma \vdash t : \triangleright A$, then the term $f \circledast t$ has type $\triangleright \left[x \leftarrow t\right] .B$, where $\left[x \leftarrow t\right]$ is a *delayed substitution*. Note that since t has type $\triangleright A$, and not A, we can not substitute t for x in B. Intuitively, t will eventually reduce to some value $\mathsf{next}\, u$, and at that time the resulting type should be $\triangleright B[u/x]$. But when t is an open term, we can not perform this reduction, and thus can not type this term. Hence we use the type mentioned earlier

$\triangleright [x \leftarrow t] .B$, in which x is bound in B. Definitional equality rules allow us to simplify this type when t has form $\mathsf{next}\, u$, i.e.,

$$\triangleright [x \leftarrow \mathsf{next}\, u] .B \simeq \triangleright B[u/x]$$

as expected. Here we have just considered a single delayed substitution, in general, we may have sequences of delayed substitutions. For example,

$$\triangleright [x \leftarrow t, y \leftarrow u] .C$$

Delayed substitutions can also occur in terms, e.g., if $\Gamma, x \colon A \vdash t : B$ and $\Gamma \vdash u : \triangleright A$, then $\Gamma \vdash \mathsf{next}\, [x \leftarrow u] .t : \triangleright [x \leftarrow u] .B$. Using this, one can express a generalisation of the rule (2.3)

$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}\, \xi.A)) \equiv \triangleright\xi.\, \mathsf{El}(A) \tag{2.9}$$

where ξ ranges over delayed substitutions. We recall the following rules from [Biz+16] which we will need in the development below. The notation $\xi\, [x \leftarrow t]$ means the extension of the delayed substitution ξ with $[x \leftarrow t]$.

$$\begin{aligned} \mathsf{next}\, \xi\, [x \leftarrow \mathsf{next}\, \xi.t] .u &= \mathsf{next}\, \xi.(u[t/x]) && \qquad (2.10)\\ \mathsf{next}\, \xi\, [x \leftarrow t] .x &= t && \qquad (2.11)\\ \mathsf{next}\, \xi\, [x \leftarrow t] .u &= \mathsf{next}\, \xi.u \qquad \text{if } x \text{ not free in } u && \qquad (2.12) \end{aligned}$$

Of these, (2.10) and (2.11) can be considered β and η laws, and (2.12) is a weakening principle.

Rather than be taken as primitive, later application $\circledast$ can be defined using delayed substitutions as

$$g \circledast y \stackrel{\text{def}}{=\!=} \mathsf{next}\, [f \leftarrow g, x \leftarrow y] .f(x)$$

Note that with this definition, the rule $\mathsf{next}(f(t)) = \mathsf{next}\, f \circledast \mathsf{next}\, t$ from Section 2.2 generalises to

$$\mathsf{next}\, \xi.(f\, t) = (\mathsf{next}\, \xi.f) \circledast (\mathsf{next}\, \xi.t)$$

which follows from (2.10).

2.5.2 Logical Relation

In this section we define a logical relation to prove the adequacy theorem. This relation is a function to $\mathcal{U}$.

We introduce the following notation:

Notation 2.22. *Let $\mathcal{R} : A \to B \to \mathcal{U}$ be a relation from A to B, t of type $\triangleright A$ and u of type $\triangleright B$. Define $t \triangleright\mathcal{R}\ u \stackrel{\text{def}}{=\!=} \triangleright [x \leftarrow t, y \leftarrow u] .(x\ \mathcal{R}\ y)$*

More precisely, we can define $t \triangleright\mathcal{R}\ u$ as a term of type $\mathcal{U}$ by defining it to be $\widehat{\triangleright}(\mathsf{next}\, [x \leftarrow t, y \leftarrow u] .(x\ \mathcal{R}\ y))$, what we have defined above are the elements of this term. From this, one can prove that

$$((\mathsf{next}\, \xi.t) \triangleright\mathcal{R}\ (\mathsf{next}\, \xi.u)) \equiv \triangleright\xi.(tRu) \tag{2.13}$$

using (2.10) and (3.21).

Lemma 2.23. *The mapping* $\lambda\,\mathcal{R}\,.\,\triangleright\mathcal{R} : (A \to B \to \mathcal{U}) \to \triangleright A \to \triangleright B \to \mathcal{U}$ *is contractive, i.e., can be factored as* $F \circ \mathsf{next}$ *for some* $F\colon \triangleright(A \to B \to \mathcal{U}) \to \triangleright A \to \triangleright B \to \mathcal{U}$.

Proof. Define $F(S)\,x\,y = \widehat{\triangleright}(S \circledast x \circledast y)$. ☐

Definition 2.24 (Logical Relation)**.** *The logical relation* $\mathcal{R}_\tau : [\![\tau]\!] \times \mathtt{Term}_{\mathtt{PCF}} \to \mathcal{U}$ *is inductively defined on types.*

$$\eta(v)\;\mathcal{R}_{\mathbf{nat}}\;M \overset{\text{def}}{=\!=} M \Downarrow^0 v$$

$$\theta_{\mathbf{nat}}(r)\;\mathcal{R}_{\mathbf{nat}}\;M \overset{\text{def}}{=\!=} \Sigma M', M''\colon \mathtt{Term}_{\mathtt{PCF}}.M \to^0_* M' \textit{ and } M' \to^1 M'' \textit{ and } r \mathrel{\triangleright\mathcal{R}_{\mathbf{nat}}} \mathsf{next}(M'')$$

$$f\;\mathcal{R}_{\tau\to\sigma}\;M \overset{\text{def}}{=\!=} \Pi\alpha\colon [\![\tau]\!], N\colon \mathtt{Term}_{\mathtt{PCF}}.\alpha\;\mathcal{R}_\tau\;N \implies f(\alpha)\;\mathcal{R}_\sigma\;(MN)$$

The definition of $\mathcal{R}_{\mathbf{nat}}$ is by guarded recursion using Lemma 2.23.

We now prove a series of lemmas needed for the proof of computational adequacy. The first states that the applicative functor action $\circledast$ respects the logical relation.

Lemma 2.25. *If* $f \mathrel{\triangleright\mathcal{R}_{\tau\to\sigma}} \mathsf{next}(M)$ *and* $r \mathrel{\triangleright\mathcal{R}_\tau} \mathsf{next}(L)$ *then*

$$(f \circledast r) \mathrel{\triangleright\mathcal{R}_\sigma} \mathsf{next}(ML)$$

Proof. The first hypothesis unfolds to

$$\triangleright\,[g \leftarrow f]\,.(g\;\mathcal{R}_{\tau\to\sigma}\;M) \simeq \triangleright\,[g \leftarrow f]\,.(\Pi(y : [\![\sigma]\!])(L : \mathtt{Term}_{\mathtt{PCF}}).y\;\mathcal{R}_\tau\;L \to g(y)\;\mathcal{R}_\sigma\;ML)$$

By delayed application of this to r, $\mathsf{next}(L)$ and the second hypothesis we get $\triangleright\,[g \leftarrow f, y \leftarrow r]\,.(g(y)\;\mathcal{R}_\sigma\;ML)$, which by (2.13) reduces to

$$\mathsf{next}\,[g \leftarrow f, y \leftarrow r]\,.(g(y)) \mathrel{\triangleright\mathcal{R}_\sigma} \mathsf{next}\,[g \leftarrow f, y \leftarrow r]\,.(ML) \simeq (f \circledast r) \mathrel{\triangleright\mathcal{R}_\sigma} \mathsf{next}(ML)$$

☐

The following lemma generalises the second case of $\mathcal{R}_{\mathbf{nat}}$ to all types.

Lemma 2.26. *Let* N *and* M *be two terms. Let* α *of type* $\triangleright[\![\sigma]\!]$, *if* $(\alpha \mathrel{\triangleright\mathcal{R}_\sigma} \mathsf{next}(N))$ *and* $M \to^1 N$ *then* $\theta_\sigma(\alpha)\;\mathcal{R}_\sigma\;M$

Proof. The proof is by induction on σ. The case $\sigma = \mathbf{nat}$ is by definition of $\mathcal{R}_{\mathbf{nat}}$.

For the induction step, suppose α of type $\triangleright[\![\tau_1 \to \tau_2]\!]$, and M, N are closed terms such that $\alpha \mathrel{\triangleright\mathcal{R}_{\tau_1\to\tau_2}} \mathsf{next}(N)$ and $M \to^1 N$. We must show that if $\beta : [\![\tau_1]\!]$, $P : \mathtt{Term}_{\mathtt{PCF}}$ and $\beta\;\mathcal{R}_{\tau_1}\;P$ then $(\theta_{\tau_1\to\tau_2}(\alpha))(\beta)\;\mathcal{R}_{\tau_2}\;(MP)$.

So suppose $\beta\;\mathcal{R}_{\tau_1}\;P$, and thus also $\triangleright(\beta\;\mathcal{R}_{\tau_1}\;P)$ which is equal to

$$\mathsf{next}(\beta) \mathrel{\triangleright\mathcal{R}_{\tau_1}} \mathsf{next}(P)$$

By applying Lemma 2.25 to this and $\alpha \mathrel{\triangleright\mathcal{R}_{\tau_1\to\tau_2}} \mathsf{next}(N)$ we get

$$\alpha \circledast (\mathsf{next}(\beta)) \mathrel{\triangleright\mathcal{R}_{\tau_2}} \mathsf{next}(NP)$$

Since $M \to^1 N$ also $MP \to^1 NP$, and thus, by the induction hypothesis for τ_2, $\theta_{\tau_2}(\alpha \circledast (\mathsf{next}(\beta)))\ \mathcal{R}_{\tau_2}\ MP$. Since by definition $\theta_{\tau_1 \to \tau_2}(\alpha)(\beta) = \theta_{\tau_2}(\alpha \circledast \mathsf{next}(\beta))$, this proves the case. ☐

Lemma 2.27. *If $M \to^0 N$ then $\alpha\ \mathcal{R}_\sigma\ M$ iff $\alpha\ \mathcal{R}_\sigma\ N$*

Proof. The proof is by induction on σ. We show the left to right implication in the case of $\sigma = \mathbf{nat}$. We proceed by case analysis on α and show the case of $\alpha = \theta_{\mathbf{nat}}(r)$. From the assumption $\alpha\ \mathcal{R}_\sigma\ M$ we have that there exists M' and M'' such that $M \to^0_* M'$ and $M' \to^1 M''$ and $\alpha \triangleright\mathcal{R}_{\mathbf{nat}}\ \mathsf{next}(M'')$. By determinism of the small-step semantics (Lemma 2.1) the reduction $M \to^0_* M'$ must factor as $M \to N \to^0_* M'$ and thus $\alpha\ \mathcal{R}_{\mathbf{nat}}\ N$ as desired. ☐

We can now finally prove the fundamental lemma, which can be thought of as a strengthened induction hypothesis for computational adequacy, generalised to open terms.

Lemma 2.28 (Fundamental Lemma)**.** *Let $\Gamma \vdash t : \tau$ and suppose $\Gamma \equiv x_1 : \tau_1, \cdots, x_n : \tau_n$. Let t_i, α_i s.t. $\alpha_i\ \mathcal{R}_{[\![\tau_i]\!]}\ t_i$ for $i \in \{1, \ldots, n\}$, then $[\![t]\!]\,(\vec{\alpha})\ \mathcal{R}_\tau\ t[\vec{t}/\vec{x}]$*

Proof. The proof is by induction on the height of the typing judgement.

The first case is when $\Gamma \vdash v : \mathbf{nat}$. We have to prove that $[\![v]\!]\,(*)\ \mathcal{R}_\tau\ v$ and – since $[\![v]\!]\,(*) = \eta(v)$ – that $v \Downarrow^0 v$. But this is true by definition of the operational semantics.

Now the variable case $\Gamma \vdash x_j : \tau_j$ where j is s.t $x_j : \tau_j \in \Gamma$ where $j \in \{1, \ldots n\}$. We want to prove that $[\![x_j]\!]\,(\vec{\alpha})\ \mathcal{R}_{\tau_j}\ x_j[\vec{t}/\vec{x}]$. By definition of substitution and by definition of the interpretation function we get $\alpha_j\ \mathcal{R}_{\tau_j}\ t_j$ which is an assumption.

The function application case is when $\Gamma \vdash t_1 t_2 : \tau$. We want to show $[\![t_1 t_2]\!]\,(\vec{\alpha})\ \mathcal{R}_\tau\ (t_1 t_2)[\vec{t}/\vec{x}]$. By induction hypothesis we have $[\![t_1]\!]\,(\vec{\alpha})\ \mathcal{R}_{\sigma \to \tau}\ (t_1)[\vec{t}/\vec{x}]$ and $[\![t_2]\!]\,(\vec{\alpha})\ \mathcal{R}_\sigma\ (t_2)[\vec{t}/\vec{x}]$. We can apply the former to the latter thus getting

$$[\![t_1]\!]\,(\vec{\alpha})\ [\![t_2]\!]\,(\vec{\alpha})\ \mathcal{R}_\tau\ (t_1[\vec{t}/\vec{x}])(t_2[\vec{t}/\vec{x}])$$

which is what we wanted to show.

Now the lambda abstraction case $\Gamma \vdash \lambda x.t : \sigma \to \tau$. We need show

$$[\![\lambda x.t]\!]\,(\vec{\alpha})\ \mathcal{R}_{\sigma \to \tau}\ (\lambda x.t)[\vec{t}/\vec{x}]$$

Following the definition of the logical relation for the lambda abstraction let $\alpha_{n+1}\ \mathcal{R}_\sigma\ t_{n+1}$. We must prove $[\![\lambda x.t]\!]\,(\vec{\alpha})(\alpha_{n+1})\ \mathcal{R}_\tau\ (\lambda x.t)[\vec{t}/\vec{x}](t_{n+1})$. By simplifying both sides, taking also into account that $x \notin \mathrm{fv}(\vec{t})$, we get

$$\begin{aligned} [\![\lambda x.t]\!]\,(\vec{\alpha})(\alpha_{n+1}) &= (\lambda x.\ [\![t]\!]\,(\vec{\alpha}))(\alpha_{n+1}) = [\![t]\!]\,(\vec{\alpha}, \alpha_{n+1}) \\ (\lambda x.t)[\vec{t}/\vec{x}](t_{n+1}) &= \lambda x.(t[\vec{t}/\vec{x}])(t_{n+1}) = (t[\vec{t}/\vec{x}])[t_{n+1}/x] \end{aligned}$$

the goal becomes $[\![t]\!]\,(\vec{\alpha}, \alpha_{n+1})\ \mathcal{R}_\tau\ t[(\vec{t}, t_{n+1})/(\vec{x}, x)]$ which is our induction hypothesis.

The most interesting case is the fixed-point case, namely $\Gamma \vdash \mathsf{Y}_\sigma\ M : \sigma$. The argument is by guarded recursion: we assume

$$\triangleright([\![\mathsf{Y}_\sigma\ M]\!]\,(\vec{\alpha})\ \mathcal{R}_\sigma\ (\mathsf{Y}_\sigma\ M)([\vec{t}/\vec{x}])) \tag{2.14}$$

and prove $[\![\mathsf{Y}_\sigma\ M]\!]\,(\vec{\alpha})\,\mathcal{R}_\sigma\,(\mathsf{Y}_\sigma\ M)([\vec{t}/\vec{x}])$. By induction hypothesis we know $[\![M]\!]\,(\vec{\alpha})\,\mathcal{R}_{\sigma\to\sigma}\,M[\vec{t}/\vec{x}]$, hence we derive $\triangleright([\![M]\!]\,(\vec{\alpha})\,\mathcal{R}_{\sigma\to\sigma}\,M[\vec{t}/\vec{x}])$, i.e.,

$$\triangleright(\Pi\alpha : [\![\sigma]\!]\,.N : \mathtt{Term}_{\mathtt{PCF}}.\ \alpha\,\mathcal{R}_\sigma\,N \Rightarrow [\![M]\!]\,(\vec{\alpha})(\alpha)\,\mathcal{R}_\sigma\ (M[\vec{t}/\vec{x}]N)) \tag{2.15}$$

Applying (2.15) to (2.14) we get

$$\triangleright([\![M]\!]\,(\vec{\alpha})([\![\mathsf{Y}_\sigma\ M]\!]\,(\vec{\alpha}))\,\mathcal{R}_\sigma\,(M[\vec{t}/\vec{x}](\mathsf{Y}_\sigma\ M[\vec{t}/\vec{x}])))$$

which is equal as types to

$$\begin{aligned}&\triangleright([\![M(\mathsf{Y}_\sigma\ M)]\!](\vec{\alpha})\,\mathcal{R}_\sigma\,(M(\mathsf{Y}_\sigma\ M))[\vec{t}/\vec{x}]\\ &\qquad\qquad\simeq \mathsf{next}([\![M(\mathsf{Y}_\sigma\ M)]\!]\,(\vec{\alpha}))\,\triangleright\mathcal{R}_\sigma\ \mathsf{next}((M(\mathsf{Y}_\sigma\ M))[\vec{t}/\vec{x}])\end{aligned}$$

Thus, by Lemma 2.26

$$\theta_\sigma(\mathsf{next}([\![M(\mathsf{Y}_\sigma\ M)(\vec{\alpha})))]\!]\ \mathcal{R}_\sigma\,(\mathsf{Y}_\sigma\ M)([\vec{t}/\vec{x}])$$

and as $\delta_\sigma = \theta_\sigma \circ \mathsf{next}$, by Lemma 2.15

$$[\![\mathsf{Y}_\sigma\ M]\!]\,(\vec{\alpha})\,\mathcal{R}_\sigma\,(\mathsf{Y}_\sigma\ M)([\vec{t}/\vec{x}])$$

as desired.

Now the case of $\Gamma \vdash \mathsf{ifz}\ L\ M\ N : \sigma$. This case can be proved by showing that

$$[\![\lambda y.\,\mathsf{ifz}\ y\ M\ N]\!]\,(\vec{\alpha})\,\mathcal{R}_{\mathbf{nat}\to\sigma}\ (\lambda y.\,\mathsf{ifz}\ y\ M\ N)[\vec{t}/\vec{x}]$$

and then applying this to the induction hypothesis $[\![L]\!]\,(\vec{\alpha})\,\mathcal{R}_{\mathbf{nat}}\,L[\vec{t}/\vec{x}]$. The argument is by guarded recursion. Assume

$$\triangleright([\![\lambda y.\,\mathsf{ifz}\ y\ M\ N]\!]\,(\vec{\alpha})\,\mathcal{R}_{\mathbf{nat}\to\sigma}\ (\lambda y.\,\mathsf{ifz}\ y\ M\ N)[\vec{t}/\vec{x}]) \tag{2.16}$$

We must show that if $\beta : [\![\mathbf{nat}]\!]$, $L : \mathtt{Term}_{\mathtt{PCF}}$ and $\beta\,\mathcal{R}_{\mathbf{nat}}\,L$ then

$$[\![\lambda y.\,\mathsf{ifz}\ y\ M\ N]\!]\,(\vec{\alpha})(\beta)\,\mathcal{R}_\sigma\,((\lambda y.\,\mathsf{ifz}\ y\ M\ N)[\vec{t}/\vec{x}](L))$$

We proceed by case analysis on β.

The first case is when $\beta = \eta(v)$. We proceed by case on v.

When $v = 0$ we get $L \Downarrow^0 0$since by assumption $\beta\,\mathcal{R}_{\mathbf{nat}}\,L$. By induction hypothesis

$$[\![M]\!]\,(\vec{\alpha})\,\mathcal{R}_\sigma\ M[\vec{t}/\vec{x}]$$

Since $v = 0$ and we know that $[\![L]\!]\,(\vec{\alpha})\,\mathcal{R}_{\mathbf{nat}}\,L[\vec{t}/\vec{x}]$ and thus that

$$\mathsf{ifz}\ L\ M[\vec{t}/\vec{x}]\ N[\vec{t}/\vec{x}] \Rightarrow^0 M[\vec{t}/\vec{x}]$$

Furthermore,

$$[\![\lambda x.\,\mathsf{ifz}\ x\ M\ N]\!]\,(\vec{\alpha})(\eta(0)) = [\![M]\!]\,(\vec{\alpha})$$

Thus by Lemma 2.27 we conclude the case.

The case for $v = k + 1$ is similar.

Now we consider $\beta = \theta_{\mathbf{nat}}(r)$. Here r is of type $\triangleright[\![\mathbf{nat}]\!]$ and $L : \mathtt{Term}_{\mathtt{PCF}}$. The hypothesis $\theta_{\mathbf{nat}}(r)\ \mathcal{R}_{\mathbf{nat}}\ L$ states that there exist $L', L'' : \mathtt{Term}_{\mathtt{PCF}}$ s.t. $L \to^0_* L'$, $L' \to^1 L''$ and

$$r\,\triangleright\mathcal{R}_{\mathbf{nat}}\ \mathsf{next}(L'') \tag{2.17}$$

Since (2.16) is equal to

$$(\mathsf{next}(\llbracket \lambda y.\, \mathsf{ifz}\ y\ M\ N \rrbracket\,(\vec{\alpha}))) \triangleright\mathcal{R}_{\mathbf{nat}\to\sigma}\ \mathsf{next}((\lambda y.\, \mathsf{ifz}\ y\ M\ N)[\vec{t}/\vec{x}])$$

We can apply Lemma 2.25 to that and (2.17) to get (using Lemma 2.27)

$$(\mathsf{next}(\llbracket \lambda y.\, \mathsf{ifz}\ y\ M\ N \rrbracket\,(\vec{\alpha})) \circledast r) \triangleright\mathcal{R}_{\sigma}\ \mathsf{next}(\mathsf{ifz}\ L''\ M[\vec{t}/\vec{x}]\ N[\vec{t}/\vec{x}])$$

By Lemma 2.26 with $L' \to^1 L''$ this implies

$$\theta_\sigma(\mathsf{next}(\llbracket \lambda y.\, \mathsf{ifz}\ y\ M\ N \rrbracket\,(\vec{\alpha})) \circledast r)\ \mathcal{R}_{\sigma}\ (\mathsf{ifz}\ L'\ M[\vec{t}/\vec{x}]\ N[\vec{t}/\vec{x}])$$

and by Lemma 2.16 along with repeated application of Lemma 2.27 this implies

$$\llbracket \lambda y.\, \mathsf{ifz}\ y\ M\ N \rrbracket\,(\vec{\alpha})(\beta)\ \mathcal{R}_{\sigma}\ \ (\lambda y.\, \mathsf{ifz}\ y\ M\ N)[\vec{t}/\vec{x}](L)$$

thus getting what we wanted. □

We have now all the pieces in place to prove adequacy.

Theorem 2.29 (Computational Adequacy). *If M is a closed term of type* **nat** *then $M \Downarrow^k v$ iff $\llbracket M \rrbracket\,(*) = \delta^k\,\llbracket v \rrbracket$.*

Proof. The left to right implication is soundness (Theorem 2.21). For the right to left implication note first that the Fundamental Lemma (Lemma 2.28) implies $\delta^k(\llbracket v \rrbracket)\ \mathcal{R}_{\mathbf{nat}}\ M$. To complete the proof it suffices to show that $\delta^k_{\mathbf{nat}}(\llbracket v \rrbracket)\ \mathcal{R}_{\mathbf{nat}}\ M$ implies $M \Downarrow^k v$.

This is proved by guarded recursion: the case of $k = 0$ is immediate by definition of $\mathcal{R}_{\mathbf{nat}}$. If $k = k' + 1$ first assume $\delta^k_{\mathbf{nat}}(\llbracket v \rrbracket)\ \mathcal{R}_{\mathbf{nat}}\ M$. By definition of $\mathcal{R}$ there exist M' and M'' such that $M \to^0_* M'$, $M' \to^1 M''$ and $\mathsf{next}(\delta^{k-1}_{\mathbf{nat}}(\llbracket v \rrbracket)) \triangleright\mathcal{R}_{\mathbf{nat}}\ \mathsf{next}(M'')$ which is type equal to $\triangleright(\delta^{k-1}_{\mathbf{nat}}(\llbracket v \rrbracket)\ \mathcal{R}_{\mathbf{nat}}\ M'')$. By the guarded recursion assumption we get $\triangleright(M'' \Downarrow^{k-1} v)$ which by Lemma 2.5 implies $M \Downarrow^k v$. □

Remark 2.30. *In the topos of trees model $\llbracket \mathbf{nat} \rrbracket\,(n) \cong \{1, \ldots, n\} \times \mathbb{N} + \{\bot\}$. Values are modelled as elements of the form $(1, k)$ and δ is defined as $\delta(j, k) = (j + 1, k)$ if $j < n$ and $\delta(n, k) = \bot$. Thus, if a term M diverges, then $\llbracket M \rrbracket\,(*) = \delta^k\,\llbracket v \rrbracket$ holds at stage n whenever $k \geq n$ explaining the need for $M \Downarrow^k v$ to be true also at stage n when $k \geq n$.*

2.6 The external viewpoint

The adequacy theorem is a statement formulated entirely in gDTT, relating two notions of semantics also formulated entirely in gDTT. While we believe that gDTT is a natural setting to do semantics in, and that the result therefore is interesting in its own right, it is still natural to ask what we proved in the “real world”. One way of formulating this question more precisely is to use the interpretation of gDTT in the topos of trees (henceforth denoted by $(\!|-|\!)$). For example, the types of PCF types, terms and values are inductively defined types, which are interpreted as constant presheaves over the corresponding *sets* of types, terms and values. Types of PCF as understood in set theory, thus correspond bijectively to global elements of $(\!|\mathtt{Type}_{\mathtt{PCF}}|\!)$, which by composing with the interpretation of PCF defined in gDTT gives rise to an object in the topos of trees. Likewise, a PCF term gives rise to a morphism in the topos of trees. Thus, essentially by composing the interpretation of PCF given above with the interpretation of gDTT, we get an interpretation of PCF into the topos of trees, which we will denote by $[\![-]\!]_{\mathtt{ext}}$.

We denote by $M \Downarrow^k_{\mathtt{ext}} v$ the usual external formulation of the big-step semantics for PCF obtained from Figure 2.0.2 by removing $\triangleright$s and replacing dependent sums by existential quantifiers (see e.g. [Esc99]).

Lemma 2.31. *The type* $(\!|M \Downarrow^k Q|\!)$ *is globally inhabited iff there exists a value* v *such that* $M \Downarrow^k_{\mathtt{ext}} v$ *and* $(\!|Q(v)|\!)$ *is globally inhabited.*

Proof. The proof is by induction over k and then M. Here we sketch the fix-point case. The object $(\!|\mathsf{Y}_\sigma\ M \Downarrow^{k+1} Q|\!)$ is globally inhabited iff $(\!|\triangleright(M(\mathsf{Y}_\sigma\ M) \Downarrow^k Q)|\!)$ is globally inhabited. Since the set of global elements of an object A is isomorphic to the set of global elements of $\triangleright A$, the latter holds iff $(\!|M(\mathsf{Y}_\sigma\ M) \Downarrow^k Q|\!)$ is globally inhabited.

By induction hypothesis, $(\!|M(\mathsf{Y}_\sigma\ M) \Downarrow^k Q|\!)$ is globally inhabited iff there exists a value v such that $M(\mathsf{Y}_\sigma\ M) \Downarrow^k_{\mathtt{ext}} v$ and $(\!|Q(v)|\!)$ is globally inhabited. The former holds iff $\mathsf{Y}_\sigma\ M \Downarrow^{k+1}_{\mathtt{ext}} v$, thus concluding the proof. □

As a special case, Theorem 2.29 states that $(\!|M \Downarrow^k v|\!)$ is inhabited by a global element iff $(\!|[\![M]\!]\,(*) = \delta^k\,[\![v]\!]|\!)$ is inhabited by a global element. Since the topos of trees is a model of extensional type theory, the latter holds precisely when $[\![M]\!]_{\mathtt{ext}} = \delta^k[\![v]\!]_{\mathtt{ext}}$.

Theorem 2.32 (Computational Adequacy, externally). *If* $\vdash M : \sigma$ *with* σ *a ground type, then* $M \Downarrow^k_{ext} v$ *iff* $[\![M]\!]_{ext}(*) = \delta^k[\![v]\!]_{ext}$

Theorem 2.32 is a restatement of Escardo's adequacy result for PCF in metric spaces [Esc99, Theorem 4.1]. Precisely, Escardo's model construction uses complete bounded ultrametric spaces. Since the spaces used are all bisected, Escardo's model can be embedded in the topos of trees [Bir+12, Section 5] and up to this embedding, his model agrees with the externalisation of the model constructed in this paper.

2.7 Discussion and Future Work

In earlier work, it has been shown how guarded type theory can be used to give abstract accounts of operationally-based step-indexed models [Bir+12; SB14]. There the operational semantics of the programming language under consideration is also defined inside guarded type theory, but there are no explicit counting of steps (indeed, part of the point is to avoid the steps). Instead, the operational semantics is defined by the transitive closure of a single-step relation — and, importantly, the transitive closure is defined by a fixed point using guarded recursion. Thus some readers might be surprised that we use a step-counting operational semantics here. The reason is simply that we want to show, in the type theory, that the denotational semantics is adequate with respect to an operational semantics and since the denotational semantics is intensional and steps thus matter, we also need to count steps in the operational semantics to formulate adequacy.

In previous work [Bir+12] we have studied the internal topos logic of the topos of trees model of guarded recursion and used this for reasoning about advanced programming languages. In this paper, we could have likewise chosen to reason in topos logic rather than type theory. We believe that the proofs of soundness and computational adequacy would have gone through also in this setting, but the interaction between the $\triangleright$ type modality and the existential quantifiers in the topos of trees, makes this an unnatural choice. For example, one can prove the statement $\exists k.\exists v.\,\mathsf{Y}_{\mathbf{nat}}\ (\lambda x.x) \Downarrow^k v$ in the internal logic using guarded recursion as follows: assume $\triangleright(\exists k.\exists v.\,\mathsf{Y}_{\mathbf{nat}}\ (\lambda x.x) \Downarrow^k v)$. Because **nat** is total and inhabited we can pull out the existentials by Theorem 2.7.4 in [Bir+12] and derive $\exists k.\exists v.\triangleright(\mathsf{Y}_{\mathbf{nat}}\ (\lambda x.x) \Downarrow^k v)$ which implies $\exists k.\exists v.\,\mathsf{Y}_{\mathbf{nat}}\ (\lambda x.x) \Downarrow^k v$. The corresponding statement in type theory: $\sum k, v.\,\mathsf{Y}_{\mathbf{nat}}\ (\lambda x.x) \Downarrow^k v$ is not derivable as can be proved using the topos of trees. Intuitively the difference is the constructiveness of the dependent sum, which allows us to extract the witnesses k and n.

In future work, we would like to explore models of FPC (i.e., PCF extended with recursive types) and also investigate how to define a more extensional model by quotienting the present intensional model. The latter would be related to Escardo's results in [Esc99].

Acknowledgements

We thank Aleš Bizjak for fruitful discussions.

Chapter 3

Denotational semantics of recursive types in synthetic guarded domain theory

Rasmus Ejlers Møgelberg and Marco Paviotti

Abstract. Guarded recursion is a form of recursion where recursive calls are guarded by delay modalities. Previous work has shown how guarded recursion is useful for reasoning operationally about programming languages with advanced features including general references, recursive types, countable non-determinism and concurrency.

Guarded recursion also offers a way of adding recursion to type theory while maintaining logical consistency. In previous work we initiated a 1 programme of denotational semantics in type theory using guarded recursion, by constructing a computationally adequate model of the language PCF (simply typed lambda calculus with fixed points). This model was intensional in that it could distinguish between computations computing the same result using a different number of fixed point unfoldings.

In this work we show how also programming languages with recursive types can be given denotational semantics in type theory with guarded recursion. More precisely, we give a computationally adequate denotational semantics to the language FPC (simply typed lambda calculus extended with recursive types), modelling recursive types using guarded recursive types. The model is intensional in the same way as was the case in previous work, but we show how to recover extensionality using a logical relation.

All constructions and reasoning in this paper, including proofs of theorems such as soundness and adequacy, are by (informal) reasoning in type theory, often using guarded recursion.

3.1 Introduction

Recent years have seen great advances in formalisation of mathematics in type theory, in particular with the development of homotopy type theory [Uni13]. Such formalisations are an important step towards machine assisted verification of mathematical proofs. Rather than adapting classical set theory based mathematics to type theory, new synthetic approaches sometimes offer simpler and clearer presentations in type theory, as illustrated by the development of synthetic homotopy theory.

Just like any other branch of mathematics, domain theory and denotational semantics for programming languages with recursion should be formalised in type theory, and, as was the case of homotopy theory, synthetic approaches can provide clearer and more abstract proofs.

Guarded recursion [Nak00] can be seen as a synthetic form of domain theory, or, perhaps more accurately, a synthetic form of step-indexing [Bir+12; App+07]. Recent work has shown how guarded recursion can be used to construct syntactic models and operational reasoning principles for (also combinations of) advanced programming language features including general references, recursive types, countable non-determinism and concurrency [Bir+12; BBM14; SB14]. The hope is that synthetic guarded domain theory can also provide denotational models of these features.

3.1.1 Synthetic guarded domain theory

The synthetic approach to domain theory is to assume that types are domains, rather than constructing a notion of domain as a type equipped with a certain structure. To model recursion a fixed point combinator is needed, but adding unrestricted fixed points makes the type theory inconsistent when read as a logical system. The approach of guarded recursion is to introduce a new type constructor $\triangleright$, pronounced "later". Elements of $\triangleright A$ are to be thought of as elements of type A available only one time step from now, and the introduction form $\mathsf{next}\colon A \to \triangleright A$ makes anything available now, available later. The fixed point operator has type

$$\mathsf{fix}\colon (\triangleright A \to A) \to A$$

and maps an f to a fixed point of $f \circ \mathsf{next}$. Guarded recursion also assumes solutions to all guarded recursive type equations, i.e., equations where all occurences of the type variable are under a $\triangleright$, as for example in the equation

$$LA \cong A + \triangleright LA \tag{3.1}$$

used to define the lifting monad L below, but guarded recursive equations can also have negative or even non-functorial occurences. Guarded recursion can be proved consistent with type theory using the topos of trees model and related variants [Bir+12; BM15; Biz+16]. In this paper we will be working in guarded dependent type theory (gDTT) [Biz+16], an extensional type theory with guarded recursion.

In previous work [PMB15], we initiated a study of denotational semantics inside guarded dependent type theory, constructing a model of PCF (simply typed lambda calculus with fixed points). By carefully aligning the fixpoint unfoldings of PCF with the steps of the metalanguage (represented by $\triangleright$), we proved a computational adequacy result for the model inside type theory. Guarded recursive types were used both in the denotational semantics (to define a lifting monad) and in the proof of computational adequacy. Likewise, the fixed point operator fix of gDTT was used both to model fixed points of PCF and as a proof principle.

3.1.2 Contributions

Here we extend our previous work in two ways. First we extend the denotational semantics and adequacy proof to languages with recursive types. More precisely, we consider the language FPC (simply typed lambda calculus extended with general recursive types), modelling recursive types using guarded recursive types. The proof of computational adequacy shows an interesting aspect of guarded domain theory. It uses a logical relation between syntax and semantics defined by induction over the structure of types. The case of recursive types requires a solution to a recursive type equation. In the setting of classical domain theory, the existence of this solution requires a separate argument [Pit96], but here it is simply a guarded recursive type.

The second contribution is a relation capturing extensionally equal elements in the model. Like the model for PCF in our previous work, the model for FPC constructed here distinguishes between programs computing the same value using a different number of fixed point unfoldings. We construct a relation on the interpretation of types, that relates elements that only differ by a finite number of computation steps. The relation is proved sound, meaning that, if the denotations of two terms are related, then the terms are contextually equivalent.

All constructions and proofs are carried out working informally in gDTT. This work illustrates the strength of gDTT, and indeed influenced the design of the type theory.

3.1.3 Related work

Escardó constructs a model of PCF using a category of ultrametric spaces [Esc99]. Since this category can be seen as a subcategory of the topos of trees [Bir+12], our previous work on PCF is a synthetic version of Escardó's model. Escardó's model also distinguishes between computations using different number of steps, and captures extensional behaviour using a logical relation similar to the one constructed here. Escardó however, does not consider recursive types. Although Escardó's model was useful for intuitions, the synthetic construction in type theory presented here is very different, in particular the proof of adequacy, which here is formulated in guarded dependent type theory.

Synthetic approaches to domain theory have been developed based on a wide range of models dating back to [Hyl91; Ros86]. Indeed, the internal languages of these models can be used to construct models of FPC and prove computational adequacy [Sim02]. A more axiomatic approach was developed in Reus's work [Reu96] where an axiomatisation of domain theory is postulated a priori inside the Extended Calculus of Constructions. Another approach is to endow the types with additional structure [BKV09; Ben+10] similar to an extensional version of the lifting monad we used in this paper. Unlike guarded synthetic domain theory, these models do not distinguish between computations using different numbers of steps. On the other hand, with the success of guarded recursion for syntactic models, we believe that the guarded approach could model languages with more advanced features.

The lifting monad used in this paper is a guarded recursive variant of the partiality monad considered by among others [Dan12; Cap05; BKV09; Ben+10]. Danielsson also defines a weak bisimulation on this monad, similar to the one defined in Definition 3.33. As reported by Danielsson, working with the partiality monad requires convincing Agda of productivity of coinductive definitions using workarounds. Here, productivity is ensured by the type system for guarded recursion.

The paper is organized as follows. Section 3.2 gives a brief introduction to the most important concepts of gDTT. More advanced constructions of the type theory are introduced as needed. Section 3.3 defines the encoding of FPC and its operational semantics in gDTT. The denotational semantics and soundness is proved in Section 3.4. Computational adequacy is proved in Section 3.5, and the relation capturing extensional equivalence is defined in Section 3.6. We conclude and discuss future work in Section 3.7.

3.2 Guarded recursion

In this paper we work informally within a type theory with dependent types, inductive types and guarded recursion. Although inductive types are not mentioned in [Biz+16] the ones used here can be safely added – as they can be modelled in the topos of trees model – and so the arguments of this paper can be formalised in gDTT. We start by recalling some core features of this theory. In fact, for the first part of the development, we will need just the features of [BM13], which corresponds to the fragment of gDTT with a single clock and no delayed substitutions. Quantification over clocks and delayed substitutions will be introduced later, when needed.

When working in type theory, we use $\equiv$ for judgemental equality of types and terms and $=$ for propositional equality (sometimes $=_A$ when we want to be explicit about the type). We also use $=$ for (external) set theoretical equality.

The type constructor $\triangleright$ introduced in Section 3.1.1 is an applicative functor in the sense of [MP08], which means that there is a map next of type $A \to \triangleright A$ and a "later application" $\circledast \colon \triangleright(A \to B) \to \triangleright A \to \triangleright B$ written infix, satisfying

$$\mathsf{next}(f) \circledast \mathsf{next}(t) \equiv \mathsf{next}(f(t)) \tag{3.2}$$

among other axioms (see also [BM13]). In particular, $\triangleright$ extends to a functor mapping $f \colon A \to B$ to $\lambda x \colon \triangleright A.\, \mathsf{next}(f) \circledast x$. Moreover, the $\triangleright$ operator distributes over the identity type as follows

$$\triangleright(t =_A u) \equiv \mathsf{next}\, t =_{\triangleright A} \mathsf{next}\, u \tag{3.3}$$

Guarded dependent type theory comes with universes in the style of Tarski. In this paper, we will just use a single universe $\mathcal{U}$. Readers familiar with [Biz+16] should think of this as $\mathcal{U}_\kappa$, but since we work with a unique clock κ, we will omit the subscript. The universe comes with codes for type operations, including $\widehat{+} \colon \mathcal{U} \times \mathcal{U} \to \mathcal{U}$ for binary sum types, codes for dependent sums and products, and $\widehat{\triangleright} \colon \triangleright\mathcal{U} \to \mathcal{U}$ satisfying $\mathsf{El}(\widehat{\triangleright}(\mathsf{next}(A))) \equiv \triangleright \mathsf{El}(A)$, where we use $\mathsf{El}(A)$ for the type corresponding to an element $A \colon \mathcal{U}$. The type of $\widehat{\triangleright}$ allows us to solve recursive type equations using the fixed point combinator. For example, if A is small, i.e., has a code $\widehat{A}$ in $\mathcal{U}$, the type equation (3.1) can be solved by computing a code of LA as

$$\mathsf{fix}(\lambda X : \triangleright\mathcal{U}.\, \widehat{+}(\widehat{A}, \widehat{\triangleright}X))\,.$$

In this paper, we will only apply the monad L to small types A.

To ease presentation, we will usually not distinguish between types and type operations on the one hand, and their codes on the other. We generally leave El implicit.

3.2.1 The topos of trees model

The topos of trees model of guarded recursion [Bir+12] provides useful intuitions, and so we briefly recall it.

In the model, a closed type is modelled as a family of sets $X(n)$ indexed by natural numbers together with restriction maps $r_n^X : X(n+1) \to X(n)$. The $\triangleright$ type operator is modelled as $\triangleright X(1) = 1, \triangleright X(n+1) = X(n)$. Intuitively, $X(n)$ is the nth approximation for computations of type X, thus $X(n)$ describes the type X as it looks if we have n computational steps to reason about it.

Using the proposition-as-types principle, types like $\triangleright^{42} 0$ are non-standard truth values. Intuitively, this is the truthvalue of propositions that appear true for 42 computation steps, but then are falsified after 43.

For guarded recursive type equations, $X(n)$ describes the nth unfolding of the type equation. For example, fixing an object A, the unique solution to (3.1) is

$$LA(n) = 1 + A(1) + \cdots + A(n)$$

with restriction maps defined using the restriction maps of A. In particular, if A is a constant presheaf, i.e., $A(n) = X$ for some fixed X and r_n^A identities, then we can think of $LA(n)$ as $\{0, \ldots, n-1\} \times X + \{\bot\}$. The set of global elements of LA is then isomorphic to $\mathbb{N} \times X + \{\bot\}$. In particular, if $X = 1$, the set of global elements is $\bar{\omega}$, the natural numbers extended with a point at infinity.

3.3 FPC

This section defines the syntax, typing judgements and operational semantics of FPC. These are inductive types in guarded type theory, but, as mentioned earlier, we work informally in type theory, and in particular remain agnostic with respect to choice of representation of syntax with binding.

Unlike the operational semantics to be defined below, the typing judgements of FPC are defined in an entirely standard way. The grammar for terms of FPC

$$\begin{aligned} L, M, N ::= {} & \langle\rangle \mid x \mid \mathtt{inl}\ M \mid \mathtt{inr}\ M \\ & \mid \mathtt{case}\ L\ \mathtt{of}\ x_1.M; x_2.N \mid \langle M, N\rangle \\ & \mid \mathtt{fst}\ M \mid \mathtt{snd}\ M \\ & \mid \lambda x : \tau.M \mid MN \mid \mathtt{fold}\ M \mid \mathtt{unfold}\ N \end{aligned}$$

should be read as an inductive type of terms in the standard way. Likewise the grammars for types and contexts and the typing judgements defined

$$\Theta \in \text{Type Contexts} \stackrel{\text{def}}{=} \langle\rangle \mid \langle\Theta, \alpha\rangle$$

$$\frac{}{\vdash \langle\rangle} \qquad \frac{\vdash \Theta}{\vdash \Theta, \alpha}\alpha \notin \Theta$$

$$\frac{\vdash \Theta}{\Theta \vdash \Theta_i} 1 \leq i \leq\mid \Theta \mid \qquad \frac{\vdash \Theta}{\Theta \vdash 1}$$

$$\frac{\Theta, \alpha \vdash \tau}{\Theta \vdash \mu\alpha.\tau} \qquad \frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1 \mathtt{op}\ \ \tau_2} \text{ for } \mathtt{op} \in \{+, \times, \to\}$$

FIGURE 3.0.1: Rules for welformed FPC types

$$\Gamma \in \text{Expression Contexts} \stackrel{\text{def}}{=} \langle\rangle \mid \langle\Gamma, x : \tau\rangle$$

$$\frac{\vdash \Theta}{\Theta \vdash \langle\rangle} \qquad \frac{\Theta \vdash \Gamma \quad \Theta \vdash \tau}{\Theta \vdash \Gamma, x : \tau} x \notin \Gamma$$

FIGURE 3.0.2: Rules for welformed FPC contexts

in Figures 3.0.1, 3.0.2 and 3.0.3 should be read as defining inductive types in type theory, allowing us to do proofs by induction over e.g. typing judgements.

We denote by $\mathtt{Type}_{\text{FPC}}$, $\mathtt{Term}_{\text{FPC}}$ and $\mathtt{Value}_{\text{FPC}}$ the types of *closed* FPC types and terms, and values of FPC. By a value we mean a closed term matching the grammar

$$v ::= \langle\rangle \mid \mathtt{inl}\ M \mid \mathtt{inr}\ M \mid \langle M, N\rangle \mid \lambda x : \tau.M \mid \mathtt{fold}\ M$$

3.3.1 Small-step semantics

Figure 3.0.4 defines the reductions of the small-step call-by-name operational semantics. Since the denotational semantics of FPC are intensional, counting reduction steps, it is necessary to also count the steps in the operational semantics in order to state the soundness and adequacy theorems precisely. More precisely, the semantics counts the number of `unfold`-`fold` reductions. The semantics is trivially deterministic.

Lemma 3.1. *The small-step semantics is deterministic: if $M \to^k N$ and $M \to^{k'} N'$, then $k = k'$ and $N = N'$.*

We next define the transitive closure of the small-step operational semantics. To ease the comparison with the big-step operational semantics, we define a generalisation of the transitive closure as a relation of the form $M \Rightarrow^k Q$ to be read as 'M reduces in k steps to a term N satisfying Q'. Here $Q\colon \mathtt{Term}_{\text{FPC}} \to \mathcal{U}$ is a (proof relevant) predicate on closed terms. The more standard big-step evaluation of terms to values can be defined as

$$M \Rightarrow^k v \stackrel{\text{def}}{=} M \Rightarrow^k (\lambda N.N = v)$$

$$\frac{x : \sigma \in \Gamma \qquad \cdot \vdash \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma . M) : \sigma \to \tau}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{inl}\, e : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{inr}\, e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash L : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash M : \sigma \quad \Gamma, x_2 : \tau_2 \vdash N : \sigma}{\Gamma \vdash \mathtt{case}\ L\ \mathtt{of}\ x_1 . M ; x_2 . N : \sigma}$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ M : \tau_1} \qquad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\, e : \tau_2}$$

$$\frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash N : \tau_2}{\Gamma \vdash \langle M, N \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash M : \mu\alpha.\tau}{\Gamma \vdash \mathtt{unfold}\, M : \tau[\mu\alpha.\tau/\alpha]} \quad \frac{\Gamma \vdash M : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathtt{fold}\, M : \mu\alpha.\tau}$$

FIGURE 3.0.3: Typing rules for FPC terms

$$(\lambda x : \sigma . M)(N) \to^0 M[N/x] \qquad \mathtt{unfold}\,(\mathtt{fold}\, M) \to^1 M$$

$$\mathtt{case}\ (\mathtt{inl}\ L)\ \mathtt{of}\ x_1 . M ; x_2 . N \to^0 M[L/x_1]$$

$$\mathtt{case}\ (\mathtt{inr}\ L)\ \mathtt{of}\ x_1 . M ; x_2 . N \to^0 N[L/x_2]$$

$$\mathtt{fst}\ \langle M, N \rangle \to^0 M \qquad \mathtt{snd}\ \langle M, N \rangle \to^0 N$$

$$\frac{M_1 \to^k M_2}{E[M_1] \to^k E[M_2]}$$

$$\begin{aligned} E ::= {} & [\cdot] \mid EM \mid \mathtt{case}\ E\ \mathtt{of}\ x_1 . M ; x_2 . N \\ & \mid \mathtt{fst}\ E \mid \mathtt{snd}\ E \mid \mathtt{unfold}\ E \end{aligned}$$

FIGURE 3.0.4: Reductions of the small-step call-by-name operational semantics. In the last rule, k is either 0 or 1.

$$
\begin{aligned}
v \Downarrow^k Q &\stackrel{\text{def}}{=} Q(v,k) \\
\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q &\stackrel{\text{def}}{=} L \Downarrow^k Q' \\
\text{where}\ \begin{array}{r} Q'(\texttt{inl}\ L, l) \\ Q'(\texttt{inr}\ L, l) \end{array} & \begin{array}{l} \stackrel{\text{def}}{=} M[L/x_1] \Downarrow^l Q \\ \stackrel{\text{def}}{=} N[L/x_2] \Downarrow^l Q \end{array} \\
\texttt{fst}\ L \Downarrow^k Q &\stackrel{\text{def}}{=} L \Downarrow^k Q' \\
\text{where } Q'(\langle M, N\rangle, m) &\stackrel{\text{def}}{=} M \Downarrow^m Q \\
\texttt{snd}\ L \Downarrow^k Q &\stackrel{\text{def}}{=} L \Downarrow^k Q' \\
\text{where } Q'(\langle M, N\rangle, m) &\stackrel{\text{def}}{=} N \Downarrow^m Q \\
MN \Downarrow^k Q &\stackrel{\text{def}}{=} M \Downarrow^k Q' \\
\text{where } Q'(\lambda x.L, m) &\stackrel{\text{def}}{=} L[N/x] \Downarrow^m Q \\
\texttt{unfold}\ M \Downarrow^k Q &\stackrel{\text{def}}{=} M \Downarrow^k Q' \\
\text{where} Q'(\texttt{fold}\ N, m+1) &\stackrel{\text{def}}{=} \triangleright(N \Downarrow^m Q)
\end{aligned}
$$

FIGURE 3.2.1: The big-step operational semantics. In the definitions of Q' only non-empty cases are given, e.g., in the case of `unfold` M, $Q'(P, n)$ is defined to be the empty type unless P is of the form `fold` N and n is a successor.

Definition 3.2. *The transitive closure of the small-step relation is defined by induction on k as follows.*

$$
\begin{gathered}
M \Rightarrow^0 Q \stackrel{\text{def}}{=} \Sigma N \colon \texttt{Term}_{\texttt{FPC}}. M \to^0_* N \text{ and } Q(N) \\
M \Rightarrow^{k+1} Q \stackrel{\text{def}}{=} \Sigma M' M'' \colon \texttt{Term}_{\texttt{FPC}}. M \to^0_* M' \text{ and} \\
M' \to^1 M'' \text{ and } \triangleright(M'' \Rightarrow^k Q)
\end{gathered}
$$

Here $\to^0_*$ is the reflexive-transitive closure of $\to^0$.

The use of $\triangleright$ in the second clause of Definition 3.2 synchronizes the steps of FPC with those of the metalogic. This allows guarded recursion to be used as a proof principle for operational semantics, and is also needed to get the precise relationship to the denotational semantics.

3.3.2 Big-step semantics

We now define a big-step call-by-name operational semantics for FPC. Big-step semantics are usually defined as relations between closed terms and values. Here, we generalize to a (proof relevant) relation of the form

$$M \Downarrow^k Q \tag{3.4}$$

where M is a term, k a natural number, and $Q \colon \texttt{Value}_{\texttt{FPC}} \times \mathbb{N} \to \mathcal{U}$ a proof relevant relation on values and natural numbers. The statement (3.4) should be read as 'M evaluates in $l \leq k$ steps to a value v such that $Q(v, k-l)$'.

As for the small-step semantics, a step is an `unfold`-`fold` reduction. If $Q\colon \texttt{Value}_{\texttt{FPC}} \to \mathcal{U}$ we overload notation and write

$$M \Downarrow^k Q \stackrel{\text{def}}{=\!=} M \Downarrow^k (\lambda \langle v, l\rangle . l = 0 \text{ and } Q(v)) \tag{3.5}$$

to be read as 'M evaluates in exactly k steps to a value satisfying Q'. We can define more standard big-step evaluation predicates as follows

$$M \Downarrow^k v \stackrel{\text{def}}{=\!=} M \Downarrow^k (\lambda w. w = v)$$
$$M \Downarrow v \stackrel{\text{def}}{=\!=} \Sigma k. M \Downarrow^k v$$

The big-step relation is defined as an inductive type in Figure 3.2.1. Following the reading of the big-step predicate given above, $MN \Downarrow^k Q$ holds if M reduces in l steps (for some $l \leq k$) to a term of the form $\lambda x.L$, such that $L[N/x] \Downarrow^{k-l} Q$. The cases of projections and `case` are similar. In the case of `unfold`, once M has been reduced to `fold` N, one time step is consumed to reduce `unfold` (`fold` N) to N before continuing reduction. Just as was the case for the small-step semantics, the use of $\triangleright$ in this rule synchronizes the steps of FPC with those of the metalogic.

The use of predicates on the right hand sides of the big-step semantics is crucial for the equivalence of the small-step and big-step semantics. More precisely, it allows us to postpone existence of terms to the time they are needed. For example, if $MN \Downarrow^k v$, and M uses one step to reduce to a value, the term $\lambda x.L$ that M should reduce to is only required to exist later, rather than now, as a more direct big-step semantics would require. This makes a difference, since Σ and $\triangleright$ do not commute.

3.3.3 Examples

As an example of a recursive type, one can encode the natural numbers as

$$\textbf{nat} \stackrel{\text{def}}{=\!=} \mu\alpha.1 + \alpha$$
$$\mathsf{zero} \stackrel{\text{def}}{=\!=} \texttt{fold}\,(\texttt{inl}\,(\langle\rangle))$$
$$\mathsf{succ}\ M \stackrel{\text{def}}{=\!=} \texttt{fold}\,(\texttt{inr}\,(M))$$

Using this definition we can define the term ifz of PCF. If L is a closed term of type **nat** and M,N are closed terms of type σ then define ifz as

$$\mathsf{ifz}\ L\ M\ N \stackrel{\text{def}}{=\!=} \texttt{case}\ (\texttt{unfold}\ L)\ \texttt{of}\ x_1.M; x_2.N$$

where x_1, x_2 are fresh. It is easy to see that $\mathsf{ifz}\ \mathsf{zero}\ M\ N \Downarrow^k Q$ iff $\triangleright(M \Downarrow^{k-1} Q)$ and that $\mathsf{ifz}\ (\mathsf{succ}\ L)\ M\ N \Downarrow^k Q$ iff $\triangleright(N \Downarrow^{k-1} Q)$ for any L closed term of type **nat**. For example, $\mathsf{ifz}\ 1\ 0\ 1 \Downarrow^2 42$ is $\triangleright 0$.

Recursive types introduce divergent terms. For example, given a type A, the Turing fixed point combinator on A can be encoded as follows:

$$
\begin{aligned}
&B \overset{\text{def}}{=\!=} \mu\alpha.(\alpha \to (A \to A) \to A) \\
&\theta : B \to (A \to A) \to A \\
&\theta \overset{\text{def}}{=\!=} \lambda x \lambda y. y(\mathtt{unfold}\, x\, x\, y) \\
&\mathsf{Y}_A \overset{\text{def}}{=\!=} \theta(\mathtt{fold}\, \theta)
\end{aligned}
$$

An easy induction shows that $\mathsf{Y}_\sigma\ (\lambda x.x) \Downarrow^k Q = \triangleright^k 0$, where 0 is the empty type.

To understand the relationship of the operational semantics defined in this paper to more traditional semantics defined without delays in the form of $\triangleright$, write $M \to_*^k N$ to mean that M reduces to N in the transitive closure of the reduction semantics, where k is the sum of the steps in the reduction. If $M \to_*^k v$ then

- $M \Downarrow^k v$ is true
- $M \Downarrow^n v$ is logically equivalent to $\triangleright^{\mathtt{min}(n,k)} 0$ if $n \neq k$, where 0 is the empty type

If, on the other hand, M is divergent in the sense that for any k there exists an N such that $M \to_*^k N$, then $M \Downarrow^n v$ is equivalent to $\triangleright^n 0$.

3.3.4 Equivalence of small-step and big-step semantics

We now state the equivalence of the two operational semantics given above. Since the big-step operational semantics as defined in (3.5) uses predicates on values, and the transitive closure of the small-step semantics (Definition 3.2) uses predicates on terms, we first introduce some notation

$$
Q_T(N) \overset{\text{def}}{=\!=} \Sigma v. N = v \text{ and } Q(v) \tag{3.6}
$$

such that $Q_T \colon \mathtt{Term}_{\text{FPC}} \to \mathcal{U}$ whenever $Q \colon \mathtt{Value}_{\text{FPC}} \to \mathcal{U}$.

Lemma 3.3. *If $M : \mathtt{Term}_{\text{FPC}}$ and $Q : \mathtt{Value}_{\text{FPC}} \to \mathcal{U}$, then $M \Downarrow^k Q$ iff $M \Rightarrow^k Q_T$.*

This has an immediate corollary.

Corollary 3.4. $M \Downarrow^k v \Leftrightarrow M \Rightarrow^k v$

Operational Correspondence We now prove the correspondence between the big-step and the small step operational semantics. First we need the following lemma.

Lemma 3.5. *Let M, N be closed terms of type τ, and let $Q : \mathtt{Value}_{\text{FPC}} \times \mathbb{N} \to \mathcal{U}$.*

1. *If $M \to^0 N$ and $N \Downarrow^k Q$ then $M \Downarrow^k Q$*
2. *If $M \to^1 N$ and $\triangleright(N \Downarrow^k Q)$ then $M \Downarrow^{k+1} Q$*

Proof. First we prove (1) by induction on $M \to^0 N$. We start off with the bases cases. When $(\lambda x.M)(N) \to^0 M[N/x]$ by assumption we have that $M[N/x] \Downarrow^k Q$, thus, the predicate Q' defined as $Q'(\lambda x.L, n) = L[N/x] \Downarrow^n Q$ – where L is bound – is true for $\lambda x.M$ and k. Hence $\lambda x.M \Downarrow^k Q'$ is also true by Definition 3.2.1 and by Definition 3.2.1 again we get $(\lambda x.M)(N) \Downarrow^k Q$. For the case of $\texttt{case}\ (\texttt{inl}\ L)\ \texttt{of}\ x_1.M; x_2.N \to^0 M[L/x_1]$ by assumption we have $M[L/x_1] \Downarrow^k Q$ and we know that the predicate Q' defined as $Q'(\texttt{inl}\ P, n) = M[P/x] \Downarrow^n Q$ – where P is bound – is true for $\texttt{inl}\ L$ and k, i.e. $Q'(\texttt{inl}\ L, k)$ is inhabited. Therefore, $\texttt{inl}\ L \Downarrow^k Q'$ is inhabited as well. It follows by Definition 3.2.1 that

$$\texttt{case}\ (\texttt{inl}\ L)\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q$$

The case of $\texttt{case}\ (\texttt{inr}\ L)\ \texttt{of}\ x_1.M; x_2.N \to^0 N[L/x_1])$ is similar to the previous case. For the case when $\texttt{fst}\ \langle M, N\rangle \to^0 M$ assume $M \Downarrow^k Q$ is inhabited, then also Q' defined as $Q'(\langle P, R\rangle, n) = P \Downarrow^n Q$ – where P, R are bound – is inhabited for $\langle M, N\rangle$ and k and thus also $\langle M, N\rangle \Downarrow^k Q'$. By Definition 3.2.1 we get $\texttt{fst}\ \langle M, N\rangle \Downarrow^k Q$. The case of $\texttt{snd}\ \langle M, N\rangle \to^0 N$ is similar to the previous case.

We now prove the inductive cases. When $M_1 N \to^0 M_2 N$ let assume $M_2 N \Downarrow^k Q$. We know by Definition 3.2.1 that $M_2 \Downarrow^k Q'$ for Q' as in Definition 3.2.1. By induction hypothesis $M_1 \Downarrow^m Q'$. Thus, by Definition 3.2.1 again we get $M_1 N \Downarrow^k Q$. For the case of $\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \to^0 \texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N$ assume

$$\texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q$$

By definition $L' \Downarrow^k Q'$. By induction hypothesis $L \Downarrow^k Q'$ and by definition $\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q$. The case when $\texttt{fst}\ M \to^0 \texttt{fst}\ M'$ assume $\texttt{fst}\ M' \Downarrow^k Q$. By definition $M' \Downarrow^k Q'$. By induction hypothesis $M \Downarrow^k Q$ which by definition is $\texttt{fst}\ M \Downarrow^k Q$. The other case $\texttt{snd}\ M \to^0 \texttt{snd}\ M'$ is similar to the previous one. When $\texttt{unfold}\ M \to^0 \texttt{unfold}\ M'$ assume $\texttt{unfold}\ M' \Downarrow^k Q$. By assumption we know that $M' \Downarrow^k Q'$ where $Q'(\texttt{fold}\ M'', n+1) = \triangleright(M'' \Downarrow^{n-1} Q)$. Since $M \to^0 M'$ by induction hypothesis $M \Downarrow^m Q'$. Thus, by definition $\texttt{unfold}\ M \Downarrow^k Q$.

We prove now (2) by induction on $M \to^1 N$. We start off with the base cases. The only case is when $\texttt{unfold}\ (\texttt{fold}\ M) \to^1 M$. So, assume $\triangleright(M \Downarrow^k Q)$ and define the predicate Q' to be

$$Q'(\texttt{fold}\ L, n+1) \overset{\text{def}}{=\!=} \triangleright L \Downarrow^{(n-1)} Q$$

It is easy to see that $Q'(\texttt{fold}\ M, k+1)$ is inhabited. Thus, by definition $\texttt{unfold}\ (\texttt{fold}\ M) \Downarrow^{k+1} Q$ holds as required.

We now prove the inductive cases. When $M_1 N \to^1 M_2 N$ assume $\triangleright(M_2 N \Downarrow^k Q)$. By definition this is equivalent to $\triangleright(M_2 \Downarrow^k Q')$ where $Q'(\lambda x.L, l) = L[N/x] \Downarrow^l Q$. By induction hypothesis we get $M_1 \Downarrow^{k+1} Q'$ which is by definition what we wanted. When $\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \to^1 \texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N$ assume

$$\triangleright(\texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q)$$

By definition $\triangleright(L' \Downarrow^k Q')$. By induction hypothesis $L \Downarrow^{k+1} Q'$ and by definition $\mathtt{case}\ L\ \mathtt{of}\ x_1.M; x_2.N \Downarrow^{k+1} Q$. When $\mathtt{fst}\ M \to^1 \mathtt{fst}\ M'$ assume $\triangleright(\mathtt{fst}\ M' \Downarrow^k Q)$. By definition $\triangleright(M' \Downarrow^k Q')$. By induction hypothesis $M \Downarrow^{k+1} Q'$. Thus, by definition $\mathtt{fst}\ M \Downarrow^{k+1} Q$. When $\mathtt{snd}\ M \to^1 \mathtt{snd}\ M'$ is similar to the previous case. When $\mathtt{unfold}\ M \to^1 \mathtt{unfold}\ M'$ assume $M \to^1 M'$ and $\triangleright(\mathtt{unfold}\ M' \Downarrow^k Q)$. By definition $\triangleright(M' \Downarrow^k Q')$ where $Q'(\mathtt{fold}\ N, m+1) = N \Downarrow^{m-1} Q$. By induction hypothesis $M \Downarrow^{k+1} Q'$ thus by definition $\mathtt{unfold}\ M \Downarrow^{k+1} Q$ as desired. ☐

The following lemma follows almost straightforwardly from the previous one.

Lemma 3.6. *For all $M N : \mathtt{Term}_{\mathrm{FPC}}$, if $M \to^0_* N$ and $N \Downarrow^k v$ then $M \Downarrow^k v$*

Proof. Trivial by using Lemma 3.5. ☐

However, we cannot prove directly Lemma 3.3 which states the correspondence between the big-step and the small-step operational semantics. This is because the right hand side of the big-step semantics in Figure 3.5 states convergence at most with some k steps whereas the small-step in Definition 3.2 – when stated with values on right hand side – states convergence in exactly some number of steps. For this reason we need an intermediate formulation of the transitive closure which we give as follows:

Definition 3.7. *Let Q be a predicate of type $\mathtt{Term}_{\mathrm{FPC}} \times \mathbb{N} \to \mathcal{U}$ and define $M \Rrightarrow^k Q$ as an inductive dependent type as follows*

$$\frac{\Sigma N \colon \mathtt{Term}_{\mathrm{FPC}}.M \to^0_* N \textit{ and } Q(N,k)}{M \Rrightarrow^k Q}$$

$$\frac{\Sigma M' M'' \colon \mathtt{Term}_{\mathrm{FPC}}.M \to^0_* M' \textit{ and } M' \to^1 M'' \textit{ and } \triangleright(M'' \Rrightarrow^k Q)}{M \Rrightarrow^{k+1} Q}$$

The following proposition follows straightforwardly from the definition.

Proposition 3.8. *For all $M, N : \mathtt{Term}_{\mathrm{FPC}}$ and $Q : \mathtt{Term}_{\mathrm{FPC}} \times \mathbb{N} \to \mathcal{U}$, if $M \to^0_* N$ and $N \Rrightarrow^k Q$ then $M \Rrightarrow^k Q$*

The small-step semantics are compositional in the following sense.

Lemma 3.9. *For all $M : \mathtt{Term}_{\mathrm{FPC}}$, if $M \Rrightarrow^k Q'$ with $Q'(L,n) = L \Rrightarrow^n Q$ then $M \Rrightarrow^k Q$*

Proof. By induction on $M \Rrightarrow^k Q'$. In the first case we have that $M \to^0_* N$ and $Q'(N,k)$, i.e. $M \Rrightarrow^k Q$, so by Proposition 3.8 we get $M \Rrightarrow^k Q$. Now the second case. By assumption we get $M \to^0_* M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rrightarrow^{k-1} Q')$. By induction hypothesis we get $\triangleright(M'' \Rrightarrow^{k-1} Q)$ which together with $M \to^0_* M'$ and $M' \to^1 M''$ give by definition $M \Rrightarrow^k Q$. ☐

The small-step semantics as in Definition 3.7 behaves well w.r.t. the contexts. To make this statement precise we define, for some context E as in Figure 3.0.4 and for some predicate Q on terms, a predicate Q_E as follows:

$$Q_E(T,k) \overset{\text{def}}{=\!=} \Sigma M.T = E[M] \text{ and } Q(M,k)$$

Intuitively, Q_E is true for terms of the form $E[M]$ for some M that satisfies Q. We use $Q_E(E[M], k)$ to explicitly indicate that the converging terms is of the form $E[M]$ and that M is the term satisfying Q. We prove now that $\Rrightarrow$ is closed under context application.

Lemma 3.10. *For all $M : \texttt{Term}_{\texttt{FPC}}$, if $M \Rrightarrow^m Q$ then $E[M] \Rrightarrow^m Q_E$.*

Proof. The proof is by induction on $M \Rrightarrow^m Q$. For the first case there exists M' such that $M \rightarrow^0_* M'$ and that satisfies the predicate $Q(M', m)$. By easy induction on the relation $\rightarrow^0_*$ we get $E[M] \rightarrow^0_* E[M']$. Now, $E[M']$ is indeed of the form required by Q_E and M' satisfies $Q(M', m)$, hence $E[M] \Rightarrow^k 0Q_E$.

As for the second case there exists m' s.t. $m = m' + 1$ and there exists an M' and M'' such that $M \rightarrow^0_* M'$, $M' \rightarrow^1 M''$ and $\triangleright(M'' \Rrightarrow^{m'} Q)$. By assumption we get $E[M] \rightarrow^0_* E[M']$ and $E[M'] \rightarrow^1 E[M'']$. By induction hypothesis we get $\triangleright(E[M''] \Rrightarrow^{m'} Q_E)$ and by definition $E[M] \Rrightarrow^{m+1} Q_E$, thus concluding the case. ☐

We prove that the small-step operational semantics is sound w.r.t the big-step.

Lemma 3.11. *Let M be a closed term and $Q : \texttt{Value}_{\texttt{FPC}} \times \mathbb{N} \rightarrow \mathcal{U}$ a relation on values. If $M \Rrightarrow^k (\lambda N.\lambda z.(N \Downarrow^z Q))$ then $M \Downarrow^k Q$*

Proof. The proof is by induction on $M \Rrightarrow^k (\lambda N.\lambda z.(N \Downarrow^z Q))$.

First case is straightforward from Lemma 3.6.

Now we prove the second case. We have that $M \rightarrow^0_* M'$, $M' \rightarrow^1 M''$ and $\triangleright(M'' \Rrightarrow^{k'} \lambda \langle N, z \rangle .(N \Downarrow^z Q))$. By induction hypothesis we know that $\triangleright(M'' \Downarrow^{k'} Q)$ and by Lemma 3.5 and Lemma 3.6 we obtain $M \Downarrow^k Q$. ☐

In the following lemma we are going to prove that the big-step operational semantics correspond to the small-step. To this end, we overload the lifting of the predicates (3.6) as follows: for a predicate $Q : \texttt{Value}_{\texttt{FPC}} \times \mathbb{N} \rightarrow \mathcal{U}$,

$$Q_T \stackrel{\text{def}}{=\!=} \lambda \langle N, k \rangle .\Sigma v.N = v \text{ and } Q(v, k)$$

Also we are going to make use of the fact that $M \Downarrow^k -$ is covariant in the following sense:

Proposition 3.12. *Let Q and R two predicates on values. If Q implies R then $M \Downarrow^k Q$ implies $M \Downarrow^k R$.*

Now we can prove that the big-step operational semantics correspond to the intermediate definition of small-step semantics.

Lemma 3.13. *If $M : \texttt{Term}_{\texttt{FPC}}$ and $Q : \texttt{Value}_{\texttt{FPC}} \times \mathbb{N} \rightarrow \mathcal{U}$, then $M \Downarrow^k Q$ iff $M \Rrightarrow^k Q_T$*

Proof. We first prove that if $M \Downarrow^k Q$ then $M \Rrightarrow^k Q_T$. The proof is by induction on $M \Downarrow^k Q$. The first case is the value. When $v \Downarrow^k Q$ by definition $Q(v, k)$ is inhabited. This together with the fact that $v \rightarrow^0_* v$ by reflexivity give us $v \Rrightarrow^k \lambda N.\lambda k.\Sigma v.N = v$ and $Q(v, k)$. When $\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q$ by definition $L \Downarrow^k Q^1$ with

$$Q^1(\texttt{inl}\ L', l) = M[L'/x] \Downarrow^l Q \text{ and } Q^1(\texttt{inr}\ L', l) = N[L'/x] \Downarrow^l Q$$

By induction hypothesis on $M[L'/x] \Downarrow^m Q$ and $N[L'/x] \Downarrow^m Q$, we know that Q^1 implies Q^2 where

$$Q^2(\texttt{inl}\ L', l) = M[L'/x] \Rrightarrow^l Q_T \text{ and } Q^2(\texttt{inr}\ L', l) = N[L'/x] \Rrightarrow^l Q_T$$

thus by Lemma 3.12 we get $L \Downarrow^k Q^2$. By induction hypothesis on $L \Downarrow^k Q^2$ we get $L \Rrightarrow^k Q^2_T$. Since Q^2 only takes values this is equivalent to $L \Rrightarrow^k Q^2$. By applying Lemma 3.10 with the context $E = \texttt{case}\ [-]\ \texttt{of}\ x_1.M; x_2.N$, we get $E[L] \Rrightarrow^k Q^2_E$ Note that the lifting of Q^2 for the context E, namely Q^2_E is equivalently formulated as

$$Q^2_E(E[\texttt{inl}\ L'], l) = M[L'/x] \Rrightarrow^l Q_T$$

and

$$Q^2_E(E[\texttt{inr}\ L'], l) = N[L'/x_1] \Rrightarrow^l Q_T$$

Since $\texttt{case inl}\ L'\ \texttt{of}\ x_1.M; x_2.N \to^0 M[L'/x_1]$ then Q^2_E is equivalently formulated as

$$Q^2_E(E[\texttt{inl}\ L'], l) = E[\texttt{inl}\ L'] \Rrightarrow^l Q_T$$

thus $\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Rrightarrow^k Q^2_E$. Directly by Lemma 3.9 we get

$$\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Rrightarrow^k Q_T$$

Now the case for the $\texttt{fst}\ L \Downarrow^k Q$. By definition $\texttt{fst}\ L \Downarrow^k Q$ is equivalent to $L \Downarrow^k Q'$ where Q' is defined as

$$Q'(\langle M, N\rangle, l) = M \Downarrow^l Q$$

By induction hypothesis on $M \Downarrow^l Q$ we get $L \Downarrow^k Q^2$ where

$$Q^2(\langle M, N\rangle, l) = M \Rrightarrow^l Q_T$$

By induction hypothesis on $L \Downarrow^k Q^2$ we get $L \Rrightarrow^k Q^2_T$ which – since Q^2 only takes values – is equivalent to $L \Rrightarrow^k Q^2$. By applying Lemma 3.10 with the context $\texttt{fst}\ [-]$, we get $\texttt{fst}\ L \Rrightarrow^k Q^3$ where Q^3 is

$$Q^3(\texttt{fst}\ \langle M, N\rangle, l) = Q^2(\langle M, N\rangle, l)$$

which can be rewritten equivalently as

$$Q^3(\texttt{fst}\ \langle M, N\rangle, l) = M \Rrightarrow^l Q_T$$

By Lemma 3.9 we get $\texttt{fst}\ L \Rrightarrow^k Q_T$ thus concluding the case. The case for $\texttt{snd}\ L \Downarrow^k Q$ is similar.

Now the case for $MN \Downarrow^k Q$. By definition $M \Downarrow^k Q^1$, where $Q^1(\lambda x.L, m) = L[N/x] \Downarrow^m Q$. By induction hypothesis on $L[N/x] \Downarrow^m Q$ we get that Q^1 implies

$$Q^2(\lambda x.L, m) = L[N/x] \Rrightarrow^m Q_T$$

thus $M \Downarrow^k Q^2$. Since $(\lambda x.L)N \to^0_* L[N/x]$, by applying Proposition 3.8 we get that Q^2 implies

$$Q^3(\lambda x.L, m) = (\lambda x.L)N \Rrightarrow^m Q_T$$

thus $M \Downarrow^k Q^3$. By applying the induction hypothesis on M we get $M \Rrightarrow^k Q^3_T$ which is equivalent to $M \Rrightarrow^k Q^3$. By applying Lemma 3.10 with context $[-]N$ we get $MN \Rrightarrow^k Q^4$ where Q^4 is defined as

$$Q^4((\lambda x.L)N, m) = Q^3(\lambda x.L, m)$$

which is equal to

$$Q^4((\lambda x.L)N, m) = (\lambda x.L)N \Rrightarrow^m Q_T$$

Directly from Lemma 3.9 we get $MN \Rrightarrow^k Q_T$.

Now the case for $\texttt{unfold}\ M \Downarrow^k Q$. By definition we have $M \Downarrow^k Q^1$ where

$$Q^1(\texttt{fold}\ N, n) = \Sigma(n-1).\triangleright(N \Downarrow^{n-1} Q)$$

By induction hypothesis on $N \Downarrow^{n-1} Q$ we get that Q^1 implies Q^2 where

$$Q^2(\texttt{fold}\ N, n) = \Sigma(n-1).\triangleright(N \Rrightarrow^{n-1} Q_T)$$

thus $M \Downarrow^k Q^2$. Since $\texttt{unfold}\ (\texttt{fold}\ N) \to^1 N$, Q^2 implies Q^3 where

$$Q^3(\texttt{fold}\ N, n) = \texttt{unfold}\ (\texttt{fold}\ N) \Rrightarrow^n Q_T$$

thus $M \Downarrow^k Q^3$. By induction on $M \Downarrow^k Q^3$ we get $M \Rrightarrow^k Q^3_T$ which is actually $M \Rrightarrow^k Q^3$. By applying Lemma 3.10 with context $\texttt{unfold}\ [-]$ we get $\texttt{unfold}\ M \Rrightarrow^k Q^4$ where

$$Q^4(\texttt{unfold}\ M', n) = Q^3(M', n)$$

which is equal to the following definition

$$Q^4(\texttt{unfold}\ \texttt{fold}\ N, n) = \texttt{unfold}\ (\texttt{fold}\ N) \Rrightarrow^n Q_T$$

Because $Q^4(N, n)$ implies $N \Rrightarrow^n Q_T$, by applying Lemma 3.9 we obtain $\texttt{unfold}\ M \Rrightarrow^k Q_T$.

We prove now that if $M \Rrightarrow^k Q_T$ then $M \Downarrow^k Q$. Assume $M \Rrightarrow^k Q_T$. Since Q_T implies Q' where

$$Q'(N, k) = N \Downarrow^k Q$$

We can apply Lemma 3.11 thus getting $M \Downarrow^k Q$. □

We now prove the two definitions for the small-step semantics coincide. To do this we have to lift the predicate on values and steps as follows: for a predicate $Q : \texttt{Term}_{\text{FPC}} \to \mathcal{U}$ define

$$Q_0(N, k) \stackrel{\text{def}}{=\!=} k = 0 \text{ and } Q(N)$$

Intuitively, Q_0 considers only reductions when the remaining number of steps is zero.

Lemma 3.14. *For all FPC terms M and $Q : \texttt{Term}_{\text{FPC}} \to \mathcal{U}$, $M \Rrightarrow^k Q_0$ iff $M \Rrightarrow^k Q$.*

Proof. We first prove the left-to-right direction by induction on $M \Rrightarrow^k Q_0$. The base case is when $M \to^0_* N$ and $Q_0(N, k)$. The latter implies $k = 0$ and

$Q(N)$, so we have to prove $M \Rightarrow^0 Q$ which is straightforward. The inductive case follows by definition.

We now prove the right-to-left direction assuming that $M \Rightarrow^k Q$ and proceeding by induction on k. The base case is $k = 0$, therefore $M \to^0_* N$ and $Q(N)$. by assumption $Q_0(N, k)$ is true. Hence $M \Rrightarrow^k Q_0$. The inductive case is $k = k' + 1$. By definition $M \to^0_* M'$ and $M' \to^1 M''$ and $\triangleright(M'' \Rightarrow^{k'} Q)$. By induction $\triangleright(M'' \Rrightarrow^{k'} Q_0)$ and by definition $M \Rrightarrow^k Q_0$. □

Now we can prove Lemma 3.3.

Proof of Lemma 3.3.

- $M \Downarrow^k Q_0$ iff $M \Rrightarrow^k Q_0$ is a particular instance of Lemma 3.13
- $M \Rrightarrow^k Q_0$ iff $M \Rightarrow^k Q$ is by Lemma 3.14

□

3.4 Denotational Semantics

We now define the denotational semantics of FPC. First we recall the definition of the guarded recursive version of the *lifting monad* on types from [PMB15]. This is defined as the *unique* solution to the guarded recursive type equation[1]

$$LA \cong A + \triangleright LA$$

which exists because the recursive variable is guarded by a $\triangleright$. This isomorphism induces a map $\theta_{LA} : \triangleright LA \to LA$ and a map $\eta \colon A \to LA$. An element of LA is either of the form $\eta(a)$ or $\theta(r)$. We think of these cases as values "now" or computations that "tick". Moreover, given $f : A \to B$ with B a $\triangleright$-algebra (i.e., equipped with a map $\theta_B \colon \triangleright B \to B$), we can lift f to a homomorphism of $\triangleright$-algebras $\hat{f} \colon LA \to B$ as follows

$$\begin{aligned} \hat{f}(\eta(a)) &\overset{\text{def}}{=\!=} f(a) \\ \hat{f}(\theta(r)) &\overset{\text{def}}{=\!=} \theta_{LB}(\mathsf{next}(\hat{f}) \circledast r) \end{aligned} \tag{3.7}$$

Formally $\hat{f}$ is defined as a fixed point of a term of type $\triangleright(LA \to B) \to LA \to B$.

Intuitively LA is the type of computations possibly returning an element of A, recording the number of steps used in the computation. We can define the divergent computation as $\bot \overset{\text{def}}{=\!=} \mathsf{fix}(\theta)$ and a "delay" map δ_{LA} of type $LA \to LA$ for any A as $\delta_{LA} \overset{\text{def}}{=\!=} \theta_{LA} \circ \mathsf{next}$. The latter can be thought of as adding a step to a computation. The lifting L extends to a functor. For a map $f : A \to B$ the action on morphisms can be defined using the unique extension as $L(f) \overset{\text{def}}{=\!=} \widehat{\eta \circ f}$.

3.4.1 Interpretation of types

The typing judgement $\Theta \vdash \tau$ is interpreted as a map of type $\mathcal{U}^{|\Theta|} \to \mathcal{U}$, where $|\Theta|$ is the cardinality of the set of variables in Θ. This interpretation map is

---

[1] Since guarded recursive types are encoded using universes, L is strictly an operation on $\mathcal{U}$. As stated in Section 3.2 we will only apply L to types that have codes in $\mathcal{U}$.

$$\begin{aligned}
\llbracket \Theta \vdash \alpha \rrbracket (\rho) &\stackrel{\text{def}}{=} \rho(\alpha) \\
\llbracket \Theta \vdash 1 \rrbracket (\rho) &\stackrel{\text{def}}{=} L1 \\
\llbracket \Theta \vdash \tau_1 \times \tau_2 \rrbracket (\rho) &\stackrel{\text{def}}{=} \llbracket \Theta \vdash \tau_1 \rrbracket (\rho) \times \llbracket \Theta \vdash \tau_2 \rrbracket (\rho) \\
\llbracket \Theta \vdash \tau_1 + \tau_2 \rrbracket (\rho) &\stackrel{\text{def}}{=} L(\llbracket \Theta \vdash \tau_1 \rrbracket (\rho) + \llbracket \Theta \vdash \tau_2 \rrbracket (\rho)) \\
\llbracket \Theta \vdash \tau_1 \to \tau_2 \rrbracket (\rho) &\stackrel{\text{def}}{=} \llbracket \Theta \vdash \tau_1 \rrbracket (\rho) \to \llbracket \Theta \vdash \tau_2 \rrbracket (\rho) \\
\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho) &\stackrel{\text{def}}{=} \triangleright(\llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)))
\end{aligned}$$

FIGURE 3.14.1: Interpretation of FPC types

defined by a combination of induction and *guarded recursion* for the case of recursive types as in Figure 3.14.1.

More precisely, the case of recursive types is defined the fixed point of a map from $\triangleright(\mathcal{U}^{|\Theta|} \to \mathcal{U})$ to $\mathcal{U}^{|\Theta|} \to \mathcal{U}$ defined as follows:

$$\lambda X.\lambda\rho.\widehat{\triangleright}(\mathsf{next}(\llbracket \tau \rrbracket) \circledast \mathsf{next}(\rho) \circledast (X \circledast \mathsf{next}(\rho)) \tag{3.8}$$

ensuring (using $\mathsf{El}(-)$ explicitly)

$$\begin{aligned}
&\mathsf{El}(\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket \rho) \\
&\equiv \mathsf{El}(\widehat{\triangleright}(\mathsf{next}(\llbracket \tau \rrbracket) \circledast \mathsf{next}(\rho) \circledast (\mathsf{next}(\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket) \circledast \mathsf{next}(\rho)))) \\
&\equiv \mathsf{El}(\widehat{\triangleright}(\mathsf{next}(\llbracket \tau \rrbracket (\rho, (\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket \rho))))) \\
&\equiv \triangleright \mathsf{El}(\llbracket \tau \rrbracket (\rho, (\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket \rho)))
\end{aligned}$$

The substitution lemma for types can be proved using guarded recursion in the case of recursive types.

Lemma 3.15 (Substitution Lemma for Types)**.** *Let σ be a well-formed type with variables in Θ and let ρ be of type $\mathcal{U}^{|\Theta|}$, for $\Theta, \beta \vdash \tau$, $\llbracket \Theta \vdash \tau[\sigma/\beta] \rrbracket (\rho) = \llbracket \Theta, \beta \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))$*

Proof. The proof is by induction on $\Theta, \beta \vdash \tau$. We start off with the case of the unit type. We know by definition that $\llbracket \Theta \vdash 1[\sigma/\beta] \rrbracket (\rho)$ is equal to $\llbracket \Theta \vdash 1 \rrbracket (\rho)$ which is equal to $\llbracket \Theta, \beta \vdash 1 \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))$, thus concluding the case. Now we consider the type variable case. Assume that $\Theta, \beta \vdash \alpha$. We split the proof in two cases. If $\alpha = \beta$ then $\llbracket \Theta \vdash \beta[\sigma/\beta] \rrbracket (\rho) = \llbracket \Theta \vdash \sigma \rrbracket (\rho)$, thus concluding the case. If α is not β then $\llbracket \Theta \vdash \alpha[\sigma/\beta] \rrbracket (\rho)$ is equal to $\llbracket \Theta \vdash \alpha \rrbracket (\rho)$ which is equal to the right-hand side.

We consider now the function case. Assume that $\Theta, \beta \vdash \sigma \to \tau$. We know by definition that $\llbracket \Theta \vdash (\tau_1 \to \tau_2)[\sigma/\beta] \rrbracket (\rho)$ is equal to

$$\llbracket \Theta \vdash \tau_1[\sigma/\beta] \to \tau_2[\sigma/\beta] \rrbracket (\rho)$$

which again by definition is equal to $\llbracket \Theta \vdash \tau_1[\sigma/\beta] \rrbracket (\rho) \to \llbracket \tau_2[\sigma/\beta] \rrbracket (\rho)$. By induction hypothesis we get

$$\llbracket \Theta \vdash \tau_1 \rrbracket (\rho, \llbracket \Gamma \vdash \sigma \rrbracket (\rho)) \to \llbracket \tau_2 \rrbracket (\rho, \llbracket \Gamma \vdash \sigma \rrbracket (\rho))$$

which by definition is $[\![\Theta \vdash \tau_1 \to \tau_2]\!]\,(\rho, [\![\Gamma \vdash \sigma]\!]\,(\rho))$. The case for $\Theta, \beta \vdash \sigma \times \tau$ is similar to the previous case.

Now we consider the case for the co-product, namely when $\Theta, \beta \vdash \tau_1 + \tau_2$. We know by definition that $[\![\Theta \vdash (\tau_1 + \tau_2)[\sigma/\beta]]\!]\,(\rho)$ equals

$$[\![\Theta \vdash \tau_1[\sigma/\beta] + \tau_2[\sigma/\beta]]\!]\,(\rho)$$

By definition of the interpretation this equals

$$L([\![\Theta \vdash \tau_1[\sigma/\beta]]\!]\,(\rho) + [\![\Theta \vdash \tau_2[\sigma/\beta]]\!]\,(\rho))$$

Now by induction hypothesis on both τ_1 and τ_2 we get

$$L([\![\Theta, \beta \vdash \tau_1]\!]\,(\rho, [\![\Gamma \vdash \sigma]\!]\,(\rho))) + L([\![\Theta, \beta \vdash \tau_2]\!]\,(\rho, [\![\Gamma \vdash \sigma]\!]\,(\rho)))$$

which is equal to $[\![\Theta, \beta \vdash \tau_1 + \tau_2]\!]\,(\rho, [\![\Gamma \vdash \sigma]\!]\,(\rho))$.

Now the most interesting case of $\Theta, \beta \vdash \mu\alpha.\tau$. We prove this case by *guarded recursion*, thus assuming that

$$\triangleright([\![\Theta \vdash (\mu\alpha.\tau)[\sigma/\beta]]\!]\,(\rho) = [\![\Theta, \beta \vdash \mu\alpha.\tau]\!]\,(\rho, [\![\Theta \vdash \sigma]\!]\,(\rho))) \tag{3.9}$$

By definition of substitution we know that $[\![\Theta \vdash \mu\alpha.\tau[\sigma/\beta]]\!]\,(\rho)$ equals

$$[\![\Theta \vdash \mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho)$$

assuming that α is not β and is not free in σ. By unfolding the definition for the denotation function on the recursive types we obtain

$$\triangleright([\![\Theta, \alpha \vdash \tau[\sigma/\beta]]\!]\,(\rho, [\![\Theta \vdash \mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho)))$$

By induction hypothesis on τ, this is equal to

$$\triangleright([\![\Theta, \alpha, \beta \vdash \tau]\!]\,(\rho, [\![\mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho), [\![\Theta, \alpha \vdash \sigma]\!]\,(\rho, [\![\mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho))))$$

Since σ is not free in α this is equal to

$$\triangleright([\![\Theta, \alpha, \beta \vdash \tau]\!]\,(\rho, [\![\mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho), [\![\Theta \vdash \sigma]\!]\,(\rho))) \tag{3.10}$$

We want now to apply the guarded recursive hypothesis. To do this note that (3.10) is a guarded fixed point construction on universes as defined in Figure 3.14.1 and explained in (3.8). Thus, (3.10) is actually equal to

$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}([\![\Theta, \alpha, \beta \vdash \tau]\!]\,(\rho, [\![\Theta \vdash \sigma]\!]\,(\rho)) \circledast (\mathsf{next}([\![\mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho)))))) \tag{3.11}$$

modulo rewritings using (3.2). Note also that $[\![\mu\alpha.(\tau[\sigma/\beta])]\!]\,(\rho)$ is appearing under the next in the form of

$$\mathsf{next}([\![\mu\alpha.(\tau[\sigma/\beta])]\!]) \circledast (\mathsf{next}(\rho)) \tag{3.12}$$

We now apply the guarded recursion hypothesis (3.9). In the hypothesis the $\triangleright$ is applied to the *equality type* (and not as a result of the fix-point construction on codes of a type). Since for every type X, Y, if $\triangleright(X = Y)$ then $\mathsf{next}(X) = \mathsf{next}(Y)$, the guarded hypothesis (3.9) implies the following equality

$$\mathsf{next}(\llbracket \Theta \vdash (\mu\alpha.\tau)[\sigma/\beta] \rrbracket (\rho)) = \mathsf{next}(\llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho)))$$

By virtue of this fact we can rewrite (3.10) as

$$\triangleright(\llbracket \Theta, \alpha, \beta \vdash \tau \rrbracket (\rho, \llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho)), \llbracket \Theta \vdash \sigma \rrbracket (\rho)))$$

By switching the order of the arguments we get

$$\triangleright(\llbracket \Theta, \beta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho), \llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))))$$

and by definition this is equal to

$$\llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))$$

thus concluding the case and the proof. □

The following lemma follows directly from the substitution lemma.

Lemma 3.16. *For all types τ and environments ρ of type $\mathcal{U}^{|\Theta|}$,*

$$\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho) = \triangleright \llbracket \Theta \vdash \tau[\mu\alpha.\tau/\alpha] \rrbracket (\rho)$$

Proof. By definition we know that $\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)$ equals to

$$\triangleright(\llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)))$$

By Substitution Lemma 3.15 this is equal to $\triangleright(\llbracket \Theta \vdash \tau[\mu\alpha.\tau/\alpha] \rrbracket (\rho))$ □

The interpretation of every *closed* type τ carries a $\triangleright$-algebra structure, i.e., a map $\theta_\tau \colon \triangleright \llbracket \tau \rrbracket \to \llbracket \tau \rrbracket$, defined by guarded recursion and structural induction on τ as in Figure 3.16.1. The case of recursive types is welltyped by Lemma 3.16. The case of products uses the functorial action of $\triangleright$ as described in Section 3.2. The case of the fixed point can be formally constructed as a fixed point of a term of type

$$G \colon \triangleright(\Pi\sigma \colon \mathtt{Type}_{\mathtt{FPC}}.(\triangleright \llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket)) \to \Pi\sigma.(\triangleright \llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket)$$

Suppose $F \colon \triangleright(\Pi\sigma \colon \mathtt{Type}_{\mathtt{FPC}}.(\triangleright \llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket))$, and define $G(F)$ essentially as in Figure 3.16.1 but with the clause $G(F)_{\tau[\mu\alpha.\tau/\alpha]}$ for recursive types being defined as

$$\lambda x \colon \triangleright \llbracket \mu\alpha.\tau \rrbracket .(F_{\tau[\mu\alpha.\tau/\alpha]} \circledast x) \tag{3.13}$$

Define θ as the fixed point of G. Then

$$\begin{aligned} \theta_{\mu\alpha.\tau}(x) &\equiv G(\mathsf{next}\,(\theta))_{\mu\alpha.\tau}(x) \\ &\equiv \mathsf{next}\,(\theta)_{\tau[\mu\alpha.\tau/\alpha]} \circledast (x) \end{aligned} \tag{3.14}$$

Using the θ we define the delay operation which, intuitively, takes a computation and adds one step.

$$\delta_\sigma \overset{\text{def}}{=\!=} \theta_\sigma \circ \mathsf{next}\,.$$

$$
\begin{aligned}
\theta_1 &\overset{\text{def}}{=\!=} \lambda x : \triangleright \llbracket 1 \rrbracket \,.\theta_{L\llbracket 1 \rrbracket}(x) \\
\theta_{\tau_1 \times \tau_2} &\overset{\text{def}}{=\!=} \lambda x : \triangleright \llbracket \tau_1 \times \tau_2 \rrbracket \,.\langle \theta_{\tau_1}(\triangleright(\pi_1)(x)), \theta_{\tau_2}(\triangleright(\pi_2)(x)) \rangle \\
\theta_{\tau_1 + \tau_2} &\overset{\text{def}}{=\!=} \lambda x : \triangleright \llbracket \tau_1 + \tau_2 \rrbracket \,.\theta_{L\llbracket \tau_1 + \tau_2 \rrbracket}(x) \\
\theta_{\sigma \to \tau} &\overset{\text{def}}{=\!=} \lambda f : \triangleright(\llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket).\lambda x : \llbracket \sigma \rrbracket \,.\theta_\tau(f \circledast (\mathsf{next}(x))) \\
\theta_{\mu\alpha.\tau} &\overset{\text{def}}{=\!=} \lambda x \colon \triangleright \llbracket \mu\alpha.\tau \rrbracket \,.\, \mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (x)
\end{aligned}
$$

FIGURE 3.16.1: Definition of $\theta_\sigma : \triangleright \llbracket \sigma \rrbracket \to \llbracket \sigma \rrbracket$

3.4.2 Interpretation of terms

Figure 3.16.2 defines the interpretation of judgements $\Gamma \vdash M : \sigma$ as functions from $\llbracket \Gamma \rrbracket$ to $\llbracket \sigma \rrbracket$ where $\llbracket x_1 : \sigma_1, \cdots, x_n : \sigma_n \rrbracket \overset{\text{def}}{=\!=} \llbracket \sigma_1 \rrbracket \times \cdots \times \llbracket \sigma_n \rrbracket$. In the case of `case` $\widehat{f}$ refers to the extension of functions to homomorphisms defined above, using the fact that all types carry a $\triangleright$-algebra structure. The interpretation of `fold` is welltyped because $\mathsf{next}(\llbracket M \rrbracket(\gamma))$ has type $\triangleright \llbracket \tau[\mu\alpha.\tau/\alpha] \rrbracket$ which by Lemma 3.16 is equal to $\llbracket \mu\alpha.\tau \rrbracket$. In the case of `unfold`, since $\llbracket M \rrbracket(\gamma)$ has type $\llbracket \mu\alpha.\tau \rrbracket$, which by Lemma 3.16 is equal to $\triangleright \llbracket \tau[\mu\alpha.\tau/\alpha] \rrbracket$, the type of $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M \rrbracket(\gamma))$ is $\llbracket \tau[\mu\alpha.\tau/\alpha] \rrbracket$.

Lemma 3.17. *For all M, if $\Gamma \vdash M : \tau[\mu\alpha.\tau/\alpha]$ then*

$$\llbracket \texttt{unfold}\,(\texttt{fold}\, M) \rrbracket(\gamma) = \delta_{\tau[\mu\alpha.\tau/\alpha]} \llbracket M \rrbracket(\gamma)$$

Proof. Straightforward by definition of the interpretation and by the type equality from Lemma 3.16. ☐

Lemma 3.18 (Substitution Lemma). *Let $\Gamma \equiv x_1 : \sigma_1, \cdots, x_k : \sigma_k$ be a context such that $\Gamma \vdash M : \tau$. For all $\Delta \vdash N_i : \sigma_i$ with $i \in \{1, \cdots, k\}$ and $\gamma \in \llbracket \Delta \rrbracket$,*

$$\left\llbracket \Delta \vdash M[\vec{N}/x] : \tau \right\rrbracket(\gamma) = \llbracket \Gamma \vdash M : \tau \rrbracket\left(\left\llbracket \Delta \vdash \vec{N} : \vec{\sigma} \right\rrbracket)(\gamma)\right)$$

Proof. By induction on the typing judgement $\Gamma \vdash M : \tau$.

The cases for $\Gamma \vdash \langle\rangle : 1$, $\Gamma \vdash x : \tau$, $\Gamma \vdash M\ N : \tau$, $\Gamma \vdash \texttt{fst}\ M : \tau_1$, $\Gamma \vdash \texttt{snd}\ M : \tau_2$, $\Gamma \vdash \langle M, N \rangle : \tau_1 \times \tau_2$ are standard.

For the case $\Gamma \vdash \texttt{inl}\ M : \tau_1 + \tau_2$ we start from

$$\left\llbracket \Delta \vdash (\texttt{inl}\ M)[\vec{N}/\vec{x}] : \tau_1 + \tau_2 \right\rrbracket(\gamma)$$

By substitution $(\texttt{inl}\ M)[\vec{N}/\vec{x}]$ equals $\texttt{inl}\ (M[\vec{N}/\vec{x}])$. We also know that its denotation equals $\eta(\mathsf{inl} \left\llbracket (M[\vec{N}/\vec{x}]) \right\rrbracket(\gamma))$ by induction hypothesis this is equal to

$$\eta(\mathsf{inl}\, \llbracket \Gamma \vdash (M) : \tau_1 + \tau_2 \rrbracket(\gamma, \left\llbracket \Delta \vdash \vec{N} : \vec{\sigma} \right\rrbracket(\gamma)))$$

which is now by definition what we wanted. The case for $\Gamma \vdash \texttt{inr}\ N : \tau_1 + \tau_2$ is similar.

$$
\begin{aligned}
\llbracket \Gamma \vdash t : \sigma \rrbracket &: \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \\
\llbracket \Gamma \vdash x \rrbracket (\gamma) &\overset{\text{def}}{=} \gamma(x) \\
\llbracket \Gamma \vdash \langle\rangle \rrbracket (\gamma) &\overset{\text{def}}{=} \eta(\star) \\
\llbracket \Gamma \vdash \langle M, N \rangle \rrbracket (\gamma) &\overset{\text{def}}{=} \langle \llbracket M \rrbracket (\gamma), \llbracket N \rrbracket (\gamma) \rangle \\
\llbracket \Gamma \vdash \texttt{fst}\ M \rrbracket (\gamma) &\overset{\text{def}}{=} \pi_1(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{snd}\ M \rrbracket (\gamma) &\overset{\text{def}}{=} \pi_2(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \lambda x.M \rrbracket (\gamma) &\overset{\text{def}}{=} \lambda x.\, \llbracket M \rrbracket (\gamma, x) \\
\llbracket \Gamma \vdash M N \rrbracket (\gamma) &\overset{\text{def}}{=} \llbracket M \rrbracket (\gamma)\, \llbracket N \rrbracket (\gamma) \\
\llbracket \Gamma \vdash \texttt{inl}\ E \rrbracket (\gamma) &\overset{\text{def}}{=} \eta(\mathsf{inl} \llbracket E \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{inr}\ E \rrbracket (\gamma) &\overset{\text{def}}{=} \eta(\mathsf{inr} \llbracket E \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \rrbracket (\gamma) &\overset{\text{def}}{=} \widehat{f}(\llbracket L \rrbracket (\gamma)) \\
\text{where } f(\mathsf{inl}(x_1)) &\overset{\text{def}}{=} \llbracket M \rrbracket (\gamma, x_1) \\
f(\mathsf{inr}(x_2)) &\overset{\text{def}}{=} \llbracket N \rrbracket (\gamma, x_2) \\
\llbracket \Gamma \vdash \texttt{fold}\ M \rrbracket (\gamma) &\overset{\text{def}}{=} \mathsf{next}(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{unfold}\ M \rrbracket (\gamma) &\overset{\text{def}}{=} \theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M \rrbracket (\gamma))
\end{aligned}
$$

FIGURE 3.16.2: Interpretation of FPC terms

Now the case for $\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \sigma$. By definition we know that $\left[\!\left[\Delta \vdash (\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N)[\vec{N}/\vec{x}] : \tau\right]\!\right](\gamma)$ is equal

$$\left[\!\left[\Delta \vdash \texttt{case}\ L[\vec{N}/\vec{x}]\ \texttt{of}\ x_1.M[\vec{N}/\vec{x}]; x_2.N[\vec{N}/\vec{x}] : \tau\right]\!\right](\gamma)$$

which is by definition of the interpretation equal to

$$\widehat{f}(\lambda x_1.\left[\!\left[M[\vec{N}/\vec{x}]\right]\!\right](\gamma, x_1), \lambda x_2.\left[\!\left[N[\vec{N}/\vec{x}]\right]\!\right](\gamma, x_2))(\left[\!\left[L[\vec{N}/\vec{x}]\right]\!\right](\gamma))$$

where $\widehat{f}$ is as in Figure 3.16.2. By induction hypothesis we know that this is equal to

$$\widehat{f}(\lambda x_1.[\![M]\!](\gamma, x_1, \left[\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\right](\gamma)), (\lambda x_2.[\![N]\!](\gamma, x_2, \left[\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\right](\gamma))))$$
$$([\![L]\!](\gamma, \left[\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\right](\gamma)))$$

which is equal by definition to

$$[\![\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \tau]\!](\gamma, \left[\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\right](\gamma))$$

Now the fixed point cases. For the case $\Gamma \vdash \texttt{unfold}\ M : \tau[\mu\alpha.\tau/\alpha]$ we know that $\left[\!\left[\Gamma \vdash (\texttt{unfold}\ M)[\vec{N}/\vec{x}]\right]\!\right](\gamma)$ is equal by definition of the substitution function to $\left[\!\left[\Gamma \vdash \texttt{unfold}\ (M[\vec{N}/\vec{x}])\right]\!\right](\gamma)$ which by definition of interpretation is $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\left[\!\left[\Gamma \vdash (M[\vec{N}/\vec{x}])\right]\!\right](\gamma))$. By induction hypothesis this is equal to

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![\Gamma \vdash M]\!](\left[\!\left[\Delta \vdash \vec{N}\right]\!\right](\gamma))$$

which by definition is $[\![\Gamma \vdash \texttt{unfold}\ (M)]\!](\left[\!\left[\Delta \vdash \vec{N}\right]\!\right](\gamma))$. For the case $\Gamma \vdash \texttt{fold}\ M : \mu\alpha.\tau$ we know that $\left[\!\left[\Gamma \vdash (\texttt{fold}\ M)[\vec{N}/\vec{x}]\right]\!\right](\gamma)$ is equal by defintion to $\left[\!\left[\Gamma \vdash \texttt{fold}\ (M[\vec{N}/\vec{x}])\right]\!\right](\gamma)$ which is by definition of the interpretation equal to $\mathsf{next}(\left[\!\left[\Gamma \vdash (M[\vec{N}/\vec{x}])\right]\!\right](\gamma))$. By induction hypothesis we get $\mathsf{fold}([\![\Gamma \vdash M]\!](\left[\!\left[\Theta \vdash \vec{N}\right]\!\right](\gamma))$ which is by definition

$$[\![\Gamma \vdash \texttt{fold}\ (M)]\!]\left[\!\left[\Theta \vdash \vec{N}\right]\!\right](\gamma)$$

□

Lemma 3.19. *1.*

$$[\![\lambda x \colon \tau_1 + \tau_2.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N]\!](\gamma)(\theta(r))$$
$$= \theta(\mathsf{next}([\![\lambda x \colon \tau_1 + \tau_2.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N]\!](\gamma)) \circledast r)$$

2. *If* $[\![L]\!](\gamma) = \delta([\![L']\!](\gamma))$, *then*

$$\begin{aligned}&[\![\texttt{case } L \texttt{ of } x_1.M; x_2.N]\!](\gamma)\\ =&\delta\left[\!\left[\texttt{case } L' \texttt{ of } x_1.M; x_2.N\right]\!\right](\gamma)\end{aligned}$$

Proof. By definition

$$[\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma)(\theta_{\tau_1+\tau_2}(\alpha))$$

is equal to

$$\widehat{f}([\![M]\!](\gamma), [\![N]\!](\gamma))(\theta_{\tau_1+\tau_2}(\alpha))$$

By definition of the unique extension for case we can pull out the θ and get

$$\theta_\sigma(\mathsf{next}(\widehat{f}([\![M]\!](\gamma), [\![N]\!](\gamma))) \circledast \alpha)$$

which is by definition of the interpretation is

$$\theta_\sigma(\mathsf{next}\,[\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma) \circledast \alpha)$$

Now we prove the second statement. By abstracting out L,

$$[\![\texttt{case } L \texttt{ of } x_1.M; x_2.N]\!](\gamma)$$

is equal to

$$[\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma)([\![L]\!](\gamma))$$

By assumption $[\![L]\!](\gamma) = \delta\,[\![L']\!](\gamma)$, this is equal to

$$[\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma)(\delta_{\tau_1+\tau_2}(\left[\!\left[L'\right]\!\right](\gamma)))$$

By definition of δ this is equal to

$$[\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma)(\theta_{\tau_1+\tau_2}(\mathsf{next}(\left[\!\left[L'\right]\!\right](\gamma))))$$

By applying the first part of this Lemma 3.19(1) we get

$$\theta_\sigma(\mathsf{next}([\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma)) \circledast (\mathsf{next}(\left[\!\left[L'\right]\!\right](\gamma))))$$

Since next distributes over $\circledast$ we get

$$\theta_\sigma(\mathsf{next}([\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma, \left[\!\left[L'\right]\!\right](\gamma))))$$

which is by defintion

$$\delta_\sigma([\![\lambda x.\texttt{case } x \texttt{ of } x_1.M; x_2.N]\!](\gamma))(\left[\!\left[L'\right]\!\right](\gamma))$$

□

Lemma 3.20. *If* $\vdash \mu\alpha.\tau$ *then*

1\.

$$[\![\ \lambda x\colon \mu\alpha.\tau.\texttt{unfold}\, x]\!](\theta_{\mu\alpha.\tau}(r)) = \theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast r)$$

2. *If* $\llbracket M \rrbracket(\gamma) = \delta_{\mu\alpha.\tau}(\llbracket M' \rrbracket(\gamma))$*, then*

$$\llbracket \mathtt{unfold}\ M \rrbracket(\gamma) = \delta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket \mathtt{unfold}\ M' \rrbracket(\gamma))$$

Proof. The interpretation for $\llbracket\, \lambda x\colon \mu\alpha.\tau.\mathtt{unfold}\ x \rrbracket\,(\theta_{\mu\alpha.\tau}(r))$ gives quite directly $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\theta_{\mu\alpha.\tau}(r))$. This type checks as r has type $\triangleright \llbracket \mu\alpha.\tau \rrbracket$, thus $(\theta_{\mu\alpha.\tau}(r))$ has type $\llbracket \mu\alpha.\tau \rrbracket$ which – by Lemma 3.16 – is equal to $\triangleright \llbracket \tau[\mu\alpha.\tau/\alpha] \rrbracket$. Thus the term $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\theta_{\mu\alpha.\tau}(r))$ has type $\llbracket \tau[\mu\alpha.\tau/\alpha] \rrbracket$. Now by definition of $\theta_{\mu\alpha.\tau}$ this is equal to $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (r))$ which is what we wanted.

Now we prove the 2) statement. By assumption and definition of δ we know that

$$\llbracket\, \lambda x\colon \mu\alpha.\tau.\mathtt{unfold}\ x \rrbracket\,(\llbracket M \rrbracket(\gamma))$$

equals to $\llbracket\, \lambda x\colon \mu\alpha.\tau.\mathtt{unfold}\ x \rrbracket\,(\theta_{\mu\alpha.\tau}(\mathsf{next}(\llbracket M' \rrbracket(\gamma))))$ Using statement 1) this is equal to $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (\mathsf{next}(\llbracket M' \rrbracket(\gamma))))$ By rule (3.2) this is equal to $\theta_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}((\theta_{\tau[\mu\alpha.\tau/\alpha]})(\llbracket M' \rrbracket(\gamma)))$ which is now by definition of the interpretation equal to

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(\llbracket\, \lambda x\colon \mu\alpha.\tau.\mathtt{unfold}\ x \rrbracket\,(\llbracket M' \rrbracket(\gamma)))$$

which by definition of δ is equal to

$$\delta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket\, \lambda x\colon \mu\alpha.\tau.\mathtt{unfold}\ x \rrbracket\,(\llbracket M' \rrbracket(\gamma)))$$

□

Lemma 3.21. *Let* M *be a closed term of type* τ*. If* $M \to^k N$ *then* $\llbracket M \rrbracket(*) = \delta^k \llbracket N \rrbracket(*)$

Proof. The proof goes by induction on $M \to^k N$. The cases when $k = 0$ follow straightforwardly from the structure of the denotational model.

The case $\mathtt{unfold}\ (\mathtt{fold}\ M) \to^1 M$ follows directly from Lemma 3.17.

The case for $(\lambda x : \sigma.M)(N) \to^0 M[N/x]$ is straightforward from by Substitution Lemma 3.18.

The case for $\mathtt{case}\ (\mathtt{inl}\ L)\ \mathtt{of}\ x_1.x.M; x_2.x.N \to^0 M[L/x]$ and the case for $\mathtt{case}\ (\mathtt{inr}\ L)\ \mathtt{of}\ x_1.x.M; x_2.x.N \to^0 N[L/x]$ follow directly by definition.

Also the elimination for the product, namely $\mathtt{fst}\ \langle M, N\rangle \to^0 M$ and $\mathtt{snd}\ \langle M, N\rangle \to^0 N$ follow directly from the definition of the interpretation.

Now we prove the inductive cases. For the case $M_1 N \to^k M_2 N$ we know that by definition $\llbracket M_1 N \rrbracket(*) = \llbracket M_1 \rrbracket(*)\llbracket N \rrbracket(*)$. By induction hypothesis we know that $\llbracket M_1 \rrbracket(*) = \delta^k_{\sigma\to\tau}(\llbracket M_2 \rrbracket(*))$, thus $\llbracket M_1 \rrbracket(*)\llbracket N \rrbracket(*) = (\delta^k_{\sigma\to\tau}(\llbracket M_2 \rrbracket(*)))\llbracket N \rrbracket(*)$ By definition of δ and θ this is equal to $\delta^k_\tau(\llbracket M_2 \rrbracket(*)\llbracket N \rrbracket(*))$.

Now the case for

$$\mathtt{case}\ L\ \mathtt{of}\ x_1.M; x_2.N \to^k \mathtt{case}\ L'\ \mathtt{of}\ x_1.M; x_2.N$$

The induction hypothesis gives $\llbracket L \rrbracket = \delta_{\tau_1+\tau_2} \circ \llbracket L' \rrbracket$, and so Lemma 3.19 applies proving the case.

The case for $\mathtt{fst}\ M \to^k \mathtt{fst}\ M'$ and for $\mathtt{snd}\ M \to^k \mathtt{snd}\ M'$ are similar to the previous case.

Finally, the case for $\mathtt{unfold}\ M_1 \to^k \mathtt{unfold}\ M_2$. By definition we know that $\llbracket \mathtt{unfold}\ M_1 \rrbracket(*) = \theta(\llbracket M_1 \rrbracket(*))$. By induction hypothesis this is equal to $\theta(\delta^k_{\mu\alpha.\tau}(\llbracket M_2 \rrbracket(*)))$ which by Lemma 3.20 is equal to $\delta^k_{\tau[\mu\alpha.\tau/\alpha]}(\theta(\llbracket M_2 \rrbracket(*)))$ thus concluding. ☐

Lemma 3.22. *Let M be a closed term of type τ, if $M \Rightarrow^k N$ then $\llbracket M \rrbracket(*) = \delta^k \llbracket N \rrbracket(*)$*

Proof. By induction on k. When $k = 0$ Lemma 3.21 applies concluding the case. When $k = n + 1$ by definition we have $M \to^0_* M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rightarrow^n N)$. By repeated application of Lemma 3.21 we get $\llbracket M \rrbracket(*) = \llbracket M' \rrbracket(*)$ and $\llbracket M' \rrbracket(*) = \delta(\llbracket M'' \rrbracket(*))$.

By induction hypothesis we get $\triangleright(\llbracket M'' \rrbracket(*) = \delta^n \llbracket N \rrbracket(*))$ which implies $\mathsf{next}(\llbracket M'' \rrbracket(*)) = \mathsf{next}(\delta^{k'} \llbracket N \rrbracket(*)))$ and since $\delta = \theta \circ \mathsf{next}$, this implies $\delta(\llbracket M'' \rrbracket(*)) = \delta^k(\llbracket N \rrbracket(*))$. By putting together the equations we get finally $\llbracket M \rrbracket(*) = \delta^k \llbracket N \rrbracket(*)$. ☐

Finally, we can prove soundness.

Theorem 3.23 (Soundness)**.** *Let M be a closed term of type τ, if $M \Downarrow^k v$ then $\llbracket M \rrbracket(*) = \delta^k \llbracket v \rrbracket(*)$*

Proof. The proof follows from the fact that the small-step semantics is equivalent to the big step, which is Corollary 3.4 and then directly by Lemma 3.22 ☐

3.5 Computational Adequacy

Computational adequacy is opposite implication of Theorem 3.23 in the case of terms of unit type. It is proved by constructing a (proof relevant) logical relation between syntax and semantics. The relation cannot be constructed just by induction on the structure of types, since in the case of recursive types, the unfolding can be bigger than the recursive type. Instead, the relation is constructed by guarded recursion: we assume the relation exists *later*, and from that assumption construct the relation *now* by structural induction on types. Thus the well-definedness of the logical relation is ensured by the type system of gDTT, more specifically by the rules for guarded recursion. This is in contrast to the classical proof in domain theory [Pit96], where existence requires a separate argument.

The logical relation uses a lifting of relations on values available now, to relations on values available later. To define this lifting, we need *delayed substitutions*, an advanced feature of gDTT.

3.5.1 Delayed substitutions

In gDTT, if $\Gamma, x \colon A \vdash B$ type is a well formed type and t has type $\triangleright A$ in context Γ, one can form the type $\triangleright [x \leftarrow t] .B$. One motivation for this is to generalise $\circledast$ (described in Section 3.2) to a dependent version: if $f \colon \triangleright(\Pi(x : A).B)$, then $f \circledast t \colon \triangleright [x \leftarrow t] .B$. The idea is that if t will eventually reduce to a term of the form $\mathsf{next}\, u$, and then $\triangleright [x \leftarrow t] .B$ will be equal to $\triangleright B[u/x]$. But if t is open, we may not be able to do this reduction yet.

More generally, we define the notion of *delayed substitution* as follows. Suppose $\Gamma, x_1 \colon A_1 \ldots x_n \colon A_n \vdash$ is a wellformed context, and all A_i are independent, i.e., no x_j appears in an A_i. A delayed substitution $\xi \colon \Gamma \rightarrow\!\triangleright$

$$\mathsf{next}\,\xi\,[x \leftarrow \mathsf{next}\,\xi.t]\,.B \equiv \mathsf{next}\,\xi.(B[t/x]) \tag{3.15}$$

$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.x \equiv t \tag{3.16}$$

$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.u \equiv \mathsf{next}\,\xi.u \tag{3.17}$$

$$\mathsf{next}\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.u \tag{3.18}$$

$$\mathsf{next}\,\xi.\,\mathsf{next}\,\xi'.u \equiv \mathsf{next}\,\xi'.\,\mathsf{next}\,\xi.u \tag{3.19}$$

$$(\mathsf{next}\,\xi.t =_{\triangleright\xi.A} \mathsf{next}\,\xi.s) \equiv \triangleright\xi.(t =_A s) \tag{3.20}$$

$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}\,\xi.A)) \equiv \triangleright\xi.\,\mathsf{El}(A) \tag{3.21}$$

FIGURE 3.23.1: The notation $\xi\,[x \leftarrow t]$ means the extension of the delayed substitution ξ with $[x \leftarrow t]$. Rule (3.17) requires x not free in u. Rule (3.19) requires that none of the variables in the codomains of ξ and ξ' appear in the type of u, and that the codomains of ξ and ξ' are independent.

$x_1 : A_1 \ldots x_n : A_n$ is a vector of terms $\xi = [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ such that $\Gamma \vdash t_i : A_i$. [Biz+16] gives a more general definition of delayed substitution allowing dependencies between the A_i's, but for this paper we just need the definition above.

If $\xi : \Gamma \rightarrowtriangle \Gamma'$ is a delayed substitution and $\Gamma, \Gamma' \vdash B$ type is a wellformed type, then the type $\triangleright\xi.B$ is wellformed in context Γ. The introduction form states $\mathsf{next}\,\xi.u : \triangleright\xi.B$ if $\Gamma, \Gamma' \vdash u : B$.

In Figure 3.23.1 we recall some rules from [Biz+16] needed below. Of these, (3.15) and (3.16) can be considered β and η laws, and (3.17) is a weakening principle. Rules (3.15), (3.17) and (3.18) also have obvious versions for types, e.g.,

$$\triangleright\xi\,[x \leftarrow \mathsf{next}\,\xi.t]\,.B \equiv \triangleright\xi.(B[t/x]) \tag{3.22}$$

Rather than be taken as primitive, later application $\circledast$ can be defined using delayed substitutions as

$$g \circledast y \overset{\text{def}}{=\!=} \mathsf{next}\,[f \leftarrow g, x \leftarrow y]\,.f(x) \tag{3.23}$$

Note that with this definition, the rule $\mathsf{next}(f(t)) \equiv \mathsf{next}\,f \circledast \mathsf{next}\,t$ from Section 3.2 generalises to

$$\mathsf{next}\,\xi.(f\,t) \equiv (\mathsf{next}\,\xi.f) \circledast (\mathsf{next}\,\xi.t) \tag{3.24}$$

which follows from (3.15).

Rules (3.16), (3.17) and (3.19) imply the rule

$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.\,\mathsf{next}\,x \equiv \mathsf{next}\,\xi\,[x \leftarrow t]\,.t$$

which by (3.20) gives an inhabitant of

$$\triangleright\xi\,[x \leftarrow t]\,.(\mathsf{next}\,x = t) \tag{3.25}$$

3.5.2 A logical relation between syntax and semantics

Figure 3.23.2 defines the logical relation between syntax and semantics. It uses the following operation lifting relations from A to B to relations from

$$
\begin{aligned}
\eta(*)\ \mathcal{R}_1\ M &\stackrel{\text{def}}{=\!=} M \Downarrow^0 \langle\rangle \\
\theta_1(x)\ \mathcal{R}_1\ M &\stackrel{\text{def}}{=\!=} \Sigma M', M'' \colon \mathtt{Term}_{\text{FPC}}.M \to^0_* M' \to^1 M'' \\
&\qquad \text{and } x \rhd\mathcal{R}_1\ \mathsf{next}(M'') \\
x\ \mathcal{R}_{\tau_1\times\tau_2}\ M &\stackrel{\text{def}}{=\!=} \pi_1(x)\ \mathcal{R}_{\tau_1}\ \mathtt{fst}\ (M) \\
&\qquad \text{and } \pi_2(x)\ \mathcal{R}_{\tau_2}\ \mathtt{snd}\ (M) \\
\eta(\mathsf{inl}(x))\ \mathcal{R}_{\tau_1+\tau_2}\ M &\stackrel{\text{def}}{=\!=} \Sigma L.M \Downarrow^0 \mathtt{inl}\ L \text{ s.t. } x\ \mathcal{R}_{\tau_1}\ L \\
\eta(\mathsf{inr}(x))\ \mathcal{R}_{\tau_1+\tau_2}\ M &\stackrel{\text{def}}{=\!=} \Sigma L.M \Downarrow^0 \mathtt{inr}\ L \text{ s.t. } x\ \mathcal{R}_{\tau_2}\ L \\
\theta_{\tau_1+\tau_2}(x)\ \mathcal{R}_{\tau_1+\tau_2}\ L &\stackrel{\text{def}}{=\!=} \Sigma M', M'' \colon \mathtt{Term}_{\text{FPC}}.M \to^0_* M' \to^1 M'' \\
&\qquad \text{and } x \rhd\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(M'') \\
f\ \mathcal{R}_{\tau\to\sigma}\ M &\stackrel{\text{def}}{=\!=} \Pi x \colon \llbracket\tau\rrbracket, N \colon \mathtt{Term}_{\text{FPC}}.x\ \mathcal{R}_\tau\ N \\
&\qquad \to f(x)\ \mathcal{R}_\sigma\ (MN) \\
x\ \mathcal{R}_{\mu\alpha.\tau}\ M &\stackrel{\text{def}}{=\!=} \Sigma M' M''.\mathtt{unfold}\ M \to^0_* M' \to^1 M'' \\
&\qquad \text{and } x \rhd\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')
\end{aligned}
$$

FIGURE 3.23.2: The logical relation $\mathcal{R}_\tau : \llbracket\tau\rrbracket \times \mathtt{Term}_{\text{FPC}} \to \mathcal{U}$.

$\rhd A$ to $\rhd B$:

$$t \rhd\mathcal{R}\ u \stackrel{\text{def}}{=\!=} \rhd\left[x \leftarrow t, y \leftarrow u\right].(x\ \mathcal{R}\ y) \tag{3.26}$$

As a consequence of (3.22) the following statement holds:

$$(\mathsf{next}\,\xi.t) \rhd\mathcal{R}\ (\mathsf{next}\,\xi.u) \equiv \rhd\xi.(t\ \mathcal{R}\ u) \tag{3.27}$$

This lifting operation can also be expressed on codes mapping $\mathcal{R} : A \to B \to \mathcal{U}$ to

$$\lambda x \colon \rhd A, y \colon \rhd B.\widehat{\rhd}(\mathsf{next}\left[x' \leftarrow x, y' \leftarrow y\right].(x'\ \mathcal{R}\ y'))$$

in fact, this operation can be shown to factor as $F \circ \mathsf{next}$, for some $F \colon \rhd(A \to B \to \mathcal{U}) \to A \to B \to \mathcal{U}$. Using this, one can formally define the logical relation as a fixed point of a function of type

$$
\begin{aligned}
&\rhd(\Pi(\tau : \mathtt{Type}_{\text{FPC}}).\, \llbracket\tau\rrbracket \times \mathtt{Term}_{\text{FPC}} \to \mathcal{U}) \to \\
&\qquad (\Pi(\tau : \mathtt{Type}_{\text{FPC}}).\, \llbracket\tau\rrbracket \times \mathtt{Term}_{\text{FPC}} \to \mathcal{U})
\end{aligned}
$$

similarly to the formal definition of θ in the equation (3.13).

3.5.3 Proof of computational adequacy

Computational adequacy follows from the fundamental lemma below, stating that all terms respect the logical relation. The proof of the fundamental lemma rests on the following two key lemmas.

Lemma 3.24. *If* $f \rhd\mathcal{R}_{\tau\to\sigma}\ \mathsf{next}(M)$ *and* $r \rhd\mathcal{R}_\tau\ \mathsf{next}(L)$ *then*

$$(f \circledast r) \rhd\mathcal{R}_\sigma\ \mathsf{next}(ML)$$

Proof. By definition $f \triangleright\mathcal{R}_{\tau\to\sigma}\ \mathsf{next}(M)$ is type equal to

$$\triangleright\,[x \leftarrow f]\,.(x\ \mathcal{R}_{\tau\to\sigma}\ M)$$

which by definition is

$$\triangleright\,[x \leftarrow f]\,.(\Pi(y : [\![\tau]\!])(L : \mathtt{Term}_{\mathtt{FPC}}).y\ \mathcal{R}_\tau\ L \to x(y)\ \mathcal{R}_\sigma\ ML)$$

By applying the latter to r we get

$$\triangleright\,[x \leftarrow f, y \leftarrow r]\,.(\Pi(L : \mathtt{Term}_{\mathtt{FPC}}).y\ \mathcal{R}_\tau\ L \to x(y)\ \mathcal{R}_\sigma\ ML)$$

By applying this to $\mathsf{next}\, L$ and the hypothesis we get

$$\triangleright\,[x \leftarrow f, y \leftarrow r]\,.(x(y)\ \mathcal{R}_\sigma\ ML)$$

which is equivalent to $(f \circledast r) \triangleright\mathcal{R}_\sigma\ \mathsf{next}(ML)$, thus concluding the case. □

Lemma 3.25. *If $x\ \mathcal{R}_\sigma\ M$ and $M \to^0 N$ then $x\ \mathcal{R}_\sigma\ N$*

Proof. We prove the statement by induction on σ. The first case is the one of the unit type. We proceed by case analysis on x. First we consider the case of $x = \eta(\star)$. From the assumption $x\ \mathcal{R}_1\ M$ we have that $M \Downarrow^0 \langle\rangle$ which by Corollary 3.4 is $M \Rightarrow^0 \langle\rangle$. This together with $M \to^0 N$ by the fact that the semantics a deterministic by Lemma 3.1 gives $N \to^0_* \langle\rangle$. Again by Corollary 3.4 we get $N \Downarrow^0 \langle\rangle$. Now the case in which $x = \theta_1(x')$. From the assumption $x\ \mathcal{R}_1\ M$ we have that there exists M' and M'' s.t. $M \to^0_* M'$ and $M' \to^1 M''$ and $x' \triangleright\mathcal{R}_1\ \mathsf{next}(M'')$. We have to prove there exist N' and N'' s.t. $N \to^0_* N'$ and $N' \to^1 N''$ and $x' \triangleright\mathcal{R}_1\ \mathsf{next}(N'')$. We set $N' = M'$ and $N'' = M''$ so that we have by assumption $N' \to^1 N''$ and $x' \triangleright\mathcal{R}_{\mathbf{nat}}\ \mathsf{next}(N'')$. We are left to prove $N \to^0_* M'$. But we have $M \to^0 N$ and $M \to^0_* M'$. This together with the fact that the operational semantics are deterministic (Lemma 3.1) implies that $N \to^0_* M'$ as we wanted.

Now we consider the case of the product. By assumption we have $\pi_1(\alpha)\ \mathcal{R}_{\tau_1}\ \mathtt{fst}\ (M)$ and $\pi_2(x)\ \mathcal{R}_{\tau_2}\ \mathtt{snd}\ (M)$. Since $M \to^0 N$ then also $\mathtt{fst}\ M \to^0 \mathtt{fst}\ N$ and $\mathtt{snd}\ M \to^0 \mathtt{snd}\ N$. By induction hypothesis on τ_1 and τ_2 we get $\pi_1(x)\ \mathcal{R}_{\tau_1}\ \mathtt{fst}\ (N)$ and $\pi_2(x))\ \mathcal{R}_{\tau_2}\ \mathtt{snd}\ (N)$. By definition of the logical relation we get what we wanted.

Now we consider the case of the coproduct. We proceed by case analysis on x. First we consider the case of $x = \eta(\mathsf{inl}(y))$. By the assumption we have that $M \Downarrow^0 \mathtt{inl}\ (N')$ and $y\ \mathcal{R}_{\tau_1}\ N'$. Since $M \to^0 N$ and the fact the operational semantics are deterministic (Lemma 3.1) we get $N \Downarrow^0 \mathtt{inl}\ N'$. By definition of the logical relation we get $x\ \mathcal{R}_{\tau_1+\tau_2}\ N$. The case where $x = \eta(\mathsf{inr}(y))$ is similar to the previous one. The interesting case is when $x = \theta_{\tau_1+\tau_2}(y)$. From the assumption $x\ \mathcal{R}_{\tau_1+\tau_2}\ M$ we have that there exists M' and M'' such that $M \to^0_* M'$ and $M' \to^1 M''$ and $y \triangleright\mathcal{R}_1\ \mathsf{next}(M'')$. We have to prove there exist N' and N'' s.t. $N \to^0_* N'$ and $N' \to^1 N''$ and $y \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(N'')$. We set $N' = M'$ and $N'' = M''$ so that we have by assumption $N' \to^1 N''$ and $y \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(N'')$. We are left to prove $N \to^0_* M'$. But we have $M \to^0 N$ and $M \to^0_* M'$. This together with the fact that the operational semantics are deterministic (Lemma 3.1) implies that $N \to^0_* M'$ as we wanted.

Now we consider the case for the function space. Assume

$$x \mathrel{\mathcal{R}_{\tau_1 \to \tau_2}} M$$

and $M \to^0 N$. Assume $y \mathrel{\mathcal{R}_{\tau_1}} P$. We have to prove $x(y) \mathrel{\mathcal{R}_{\tau_2}} NP$. By induction hypothesis we know that for all x, M, N, $x \mathrel{\mathcal{R}_{\tau_2}} M$ and $M \to^0 N$ then $x \mathrel{\mathcal{R}_{\tau_2}} N$. By applying it we have left to show that $MP \to^0 NP$ which is trivial and $x(y) \mathrel{\mathcal{R}_{\tau_2}} MP$. The latter we get by using $y \mathrel{\mathcal{R}_{\tau_1}} P$ with the hypothesis $x \mathrel{\mathcal{R}_{\tau_1 \to \tau_2}} M$ thus concluding the proof.

Now we consider the case for recursive types. By assumption we know there exists M' and M'' such that $\mathtt{unfold}\ M \to^0_* M'$ and $M' \to^1 M''$ and $x \mathrel{\triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}} \mathsf{next}(M'')$. Since $M \to^0 N$ then also $\mathtt{unfold}\ M \to^0 \mathtt{unfold}\ N$. Therefore, from the assumption and the fact that the operational semantics are deterministic (Lemma 3.1) we get $\mathtt{unfold}\ N \to^0_* M'$. By definition of the logical relation we get $x \mathrel{\mathcal{R}_{\mu\alpha.\tau}} N$, which concludes the proof. □

Lemma 3.26. *For all $x \mathrel{\mathcal{R}_\sigma} N$ and $M \to^0 N$ then $x \mathrel{\mathcal{R}_\sigma} M$*

Proof. We prove the statement by induction on σ. The first case is the case of the unit type. We do it by case analysis on x. When $x = \eta(\star)$ we have that $N \Downarrow^0 \langle\rangle$ which by Corollary 3.4 $N \to^0_* \langle\rangle$. Since $M \to^0 N$ we get $M \to^0_* \langle\rangle$. Again by Corollary 3.4 we get $M \Downarrow^0 \langle\rangle$ and by definition of the logical relation we conclude the case. When x is $\theta_1(x')$ by assumption $x \mathrel{\mathcal{R}_1} N$ implies that there exists N' and N'' such that $N \to^0_* N'$ and $N' \to^1 N''$ and $x \mathrel{\triangleright\mathcal{R}_1} \mathsf{next}(N'')$. We have to prove there exist M' and M'' such that $M \to^0_* M'$ and $M' \to^1 M''$ and $x \mathrel{\triangleright\mathcal{R}_1} \mathsf{next}(M'')$. We pick $M' = N'$ and $M'' = N''$ so that we have by assumption $M' \to^1 M''$ and $x \mathrel{\triangleright\mathcal{R}_1} \mathsf{next}(M'')$. We are left to prove $M \to^0_* N'$, but we have $M \to^0 N$ and $N \to^0_* N'$ so $M \to^0_* N'$ is by definition. By definition of the logical relation we conclude.

Now we consider the case for $\tau_1 \times \tau_2$. By assumption we have that $\pi_1(x) \mathrel{\mathcal{R}_{\tau_1}} \mathtt{fst}\ N$ and $\pi_2(x) \mathrel{\mathcal{R}_{\tau_2}} \mathtt{snd}\ N$. Since $M \to^0 N$ then also $\mathtt{fst}\ (M) \to^0 \mathtt{fst}\ (N)$ $\mathtt{snd}\ (M) \to^0 \mathtt{snd}\ (N)$. By induction hypothesis on both τ_1 and τ_2 we get $\pi_1(x) \mathrel{\mathcal{R}_{\tau_1}} \mathtt{fst}\ (M)$ and $\pi_1(x) \mathrel{\mathcal{R}_{\tau_1}} \mathtt{snd}\ (M)$. By definition of the logical relation we get what we wanted.

Now we consider the co-product case. We proceed by case analysis on x. When $x = \eta(\mathsf{inl}(y))$ we have that $N \Downarrow^0 \mathtt{inl}\ (N')$ and $y \mathrel{\mathcal{R}_{\tau_1}} N'$. By Corollary 3.4 we derive that $N \to^0_* \mathtt{inl}\ (N')$ Since $M \to^0 N$ we get $M \to^0_* \mathtt{inl}\ (N')$. Again by Corollary 3.4 we get $M \Downarrow^0 \mathtt{inl}\ (N')$. By definition of the logical relation we get $x \mathrel{\mathcal{R}_{\tau_1+\tau_2}} M$. The case for $x = \eta(\mathsf{inr}(x'))$ is similar to the previous one.

Now we consider the case when $x = \theta_{\tau_1+\tau_2}(x')$. From the assumption $x \mathrel{\mathcal{R}_{\tau_1+\tau_2}} N$ we have that there exists N' and N'' such that $N \to^0_* N'$ and $N' \to^1 N''$ and $x \mathrel{\triangleright\mathcal{R}_{\tau_1+\tau_2}} \mathsf{next}(N'')$. We have to prove there exist M' and M'' such that $M \to^0_* M'$ and $M' \to^1 M''$ and $x \mathrel{\triangleright\mathcal{R}_{\tau_1+\tau_2}} \mathsf{next}(M'')$. We pick $M' = N'$ and $M'' = N''$ so that we have by assumption $M' \to^1 M''$ and $x \mathrel{\triangleright\mathcal{R}_{\tau_1+\tau_2}} \mathsf{next}(M'')$. We are left to prove $M \to^0_* N'$, but we have $M \to^0 N$ and $N \to^0_* N'$ so $M \to^0_* N'$ is by definition. By definition of the logical relation we conclude. Now we consider the function space. Assume $x \mathrel{\mathcal{R}_{\tau_1 \to \tau_2}} N$ and $M \to^0 N$. Assume $y \mathrel{\mathcal{R}_{\tau_1}} P$. We have to prove $x(y) \mathrel{\mathcal{R}_{\tau_2}} MP$. By induction hypothesis we know that for all x, M, N, $x \mathrel{\mathcal{R}_{\tau_2}} N$ and $M \to^0 N$ then $x \mathrel{\mathcal{R}_{\tau_2}} M$. By applying it to the goal we have left to show that $MP \to^0 NP$ which is trivial and $x(y) \mathrel{\mathcal{R}_{\tau_2}} NP$. The latter we get by using $y\ \mathcal{R}_{\tau_1}\ P$ with the hypothesis $x\ \mathcal{R}_{\tau_1 \to \tau_2}\ N$ thus concluding the proof. Now we consider the recursive types. By assumption we have that $x\ \mathcal{R}_{\mu\alpha.\tau}\ N$ and $M \to^0 N$. From the former we derive that there exists M' and M'' such that `unfold` $N \to^0_* M'$,$M' \to^1 M''$ and $x\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$. Since $M \to^0 N$ then also `unfold` $M \to^0$ `unfold` N. Therefore, we know that `unfold` $M \to^0_* M'$, thus by definition of the logical relation we conclude. □

Lemma 3.27. *If $M \to^0_* N$ then $x\ \mathcal{R}_\sigma\ M$ iff $x\ \mathcal{R}_\sigma\ N$.*

Proof. We prove the statement by induction $M \to^0_* N$. The case where $M = N$ is straightforwardly true. Now we consider the case for $M \to^0 N$. The implication from left to right is direct from Lemma 3.25 and the other by Lemma 3.26. Now we consider the case when $M \to^0 M'$ and $M' \to^0_* N$. First we prove the left-to-right implication. By Lemma 3.25 we have $x\ \mathcal{R}_\sigma\ M'$ and by induction hypothesis we get $x\ \mathcal{R}_\sigma\ N$. The right-to-left case mirrors the previous one by applying Lemma 3.26 instead. □

Lemma 3.28. *If $x \triangleright\mathcal{R}_\tau\ \mathsf{next}(M)$ and $M' \to^1 M$ then $\theta_\tau(x)\ \mathcal{R}_\tau\ M'$.*

Proof. The proof is by guarded recursion, so we assume that the lemma is "later true", i.e., that we have an inhabitant of the type obtained by applying $\triangleright$ to the statement of the lemma. We proceed by induction on τ.

The case for the unit type and for the co-product is straightforward by definition. Now the case for the product. By assumption we have

$$\alpha \triangleright\mathcal{R}_{\tau_1 \times \tau_2}\ \mathsf{next}(M)$$

by pulling out the $\triangleright$ and unfolding the definition of the logical relation we get

$$\triangleright[x \leftarrow \alpha].(\pi_1(x)\ \mathcal{R}_{\tau_1}\ (\texttt{fst}\ M)) \text{ and } (\pi_2(x)\ \mathcal{R}_{\tau_2}\ \texttt{fst}\ (M))$$

Since the$\triangleright$ distributes over products we derive

$$(\pi_1(\alpha)) \triangleright\mathcal{R}_{\tau_1}\ \mathsf{next}(\texttt{fst}\ M) \qquad \text{and} \qquad \pi_2(\alpha) \triangleright\mathcal{R}_{\tau_2}\ \mathsf{next}(\texttt{snd}\ M)$$

Since $M' \to^1 M$ then also `fst` $M' \to^1$ `fst` M and `snd` $M' \to^1$ `snd` M, thus we can use the induction hypothesis on τ_1 and τ_2 and get

$$\theta_{\tau_1}(\pi_1(\alpha))\ \mathcal{R}_{\tau_1}\ \texttt{fst}\ M' \qquad \text{and} \qquad \theta_{\tau_2}(\pi_2(\alpha))\ \mathcal{R}_{\tau_2}\ \texttt{snd}\ M'$$

by definition $\theta_{\tau_1 \times \tau_2}$ commutes with π_1 and π_2. Thus, we obtain

$$\pi_1(\theta_{\tau_1 \times \tau_2}(\alpha))\ \mathcal{R}_{\tau_1}\ \texttt{fst}\ M' \qquad \text{and} \qquad \pi_2(\theta_{\tau_1 \times \tau_2}(\alpha))\ \mathcal{R}_{\tau_2}\ \texttt{snd}\ M'$$

which is by definition what we wanted.

Now the case for the function space. Assume $\alpha \triangleright\mathcal{R}_{\tau_1 \to \tau_2}\ \mathsf{next}(M)$ and $M' \to^1 M$. We must show that if $\beta : [\![\tau_1]\!]$, $N : \texttt{Term}_{\texttt{FPC}}$ and $\beta\ \mathcal{R}_{\tau_1}\ N$ then $(\theta_{\tau_1 \to \tau_2}(\alpha))(\beta)\ \mathcal{R}_{\tau_2}\ (MN)$. So suppose $\beta\ \mathcal{R}_{\tau_1}\ N$, and thus also $\triangleright(\beta\ \mathcal{R}_{\tau_1}\ N)$ which is equal to $\mathsf{next}(\beta) \triangleright\mathcal{R}_{\tau_1}\ \mathsf{next}(N)$. By applying Lemma 3.24 to this and

$\alpha \triangleright\mathcal{R}_{\tau_1 \to \tau_2}\ \mathsf{next}(M)$ we get

$$\alpha \circledast (\mathsf{next}(\beta)) \triangleright\mathcal{R}_{\tau_2}\ \mathsf{next}(MN)$$

Since $M' \to^1 M$ also $M'N \to^1 MN$, and thus, by the induction hypothesis for τ_2, $\theta_{\tau_2}(\alpha \circledast (\mathsf{next}(\beta)))\ \mathcal{R}_{\tau_2}\ M'N$. Since by definition $\theta_{\tau_1 \to \tau_2}(\alpha)(\beta) = \theta_{\tau_2}(\alpha \circledast \mathsf{next}(\beta))$, this proves the case.

The interesting case is the one of $\mu\alpha.\tau$. Assume $x \triangleright\mathcal{R}_{\mu\alpha.\tau}\ \mathsf{next}(M)$ and $M' \to^1 M$. By definition of $\triangleright\mathcal{R}$ this implies $\triangleright [y \leftarrow x] .(y\ \mathcal{R}_{\mu\alpha.\tau}\ M)$ which by definition of $\mathcal{R}_{\mu\alpha.\tau}$ is

$$\begin{aligned}\triangleright [y \leftarrow x] .\Sigma N' N''.\texttt{unfold}\ M \to^0_* N' \text{ and}\\ N' \to^1 N'' \text{ and } (y \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(N''))\end{aligned}$$

Since zero-step reductions cannot eliminate outer `unfold`'s, N' must be on the form $\texttt{unfold}\ N$ for some N, such that $M \to^0_* N$. Thus, we can apply the guarded induction hypothesis to get

$$\begin{aligned}\triangleright [y \leftarrow x] .(\Sigma N.M \to^0_* N \text{ and}\\ (\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \texttt{unfold}\ N))\end{aligned}$$

Since $\texttt{unfold}\ M \to^0_* \texttt{unfold}\ N$, by Lemma 3.27 we get

$$\triangleright [y \leftarrow x] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \texttt{unfold}\ M)$$

which by (3.27) is

$$\mathsf{next}\ [y \leftarrow x] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)) \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(\texttt{unfold}\ M)$$

By (3.23) this implies

$$\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast x \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(\texttt{unfold}\ M)$$

Since by assumption $M' \to^1 M$ also $\texttt{unfold}\ M' \to^1 \texttt{unfold}\ M$ thus, by definition of the logical relation

$$\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast x\ \mathcal{R}_{\mu\alpha.\tau}\ M'$$

By definition $\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast x$ is equal to $\theta_{\mu\alpha.\tau}(x)$ thus we can derive

$$\theta_{\mu\alpha.\tau}(x)\ \mathcal{R}_{\mu\alpha.\tau}\ M'$$

as we wanted. □

Lemma 3.29 (Fundamental Lemma). *Suppose* $\Gamma \vdash M : \tau$, *for* $\Gamma \equiv x_1 : \tau_1, \cdots, x_n : \tau_n$ *and* $N_i : \tau_i$, $\gamma_i : [\![\tau_i]\!]$ *and* $\gamma_i\ \mathcal{R}_{[\![\tau_i]\!]}\ N_i$ *for* $i \in \{1, \ldots, n\}$, *then* $[\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_\tau\ M[\vec{N}/\vec{x}]$

Proof. We assume by guarded recursion that for all well-typed terms M with context Γ and type τ the following holds:

$$\triangleright([\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_\tau\ M[\vec{M}/\vec{x}])$$

Then we proceed by induction on the typing derivation $\Gamma \vdash M : \tau$. For the case $\Gamma \vdash \langle\rangle : 1$, we know by definition of the logical relation that $[\![\langle\rangle]\!]\,(\vec{\gamma})\ \mathcal{R}_1\ \langle\rangle[\vec{M}/\vec{x}]$ is equal to $\langle\rangle \Downarrow^0 \langle\rangle$ which is true by definition of the big-step operational semantics. Now the case $\Gamma \vdash x_j : \tau_j$. Here j is s.t $x_j : \tau_j \in \Gamma$ where $j \in \{1, \ldots n\}$ and $\gamma_j\ \mathcal{R}_\tau\ M_j$. We want to prove that $[\![x_j]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_j}\ x_j[\vec{M}/\vec{x}]$. By the substitution function $x_j[\vec{M}/\vec{x}] = M_j$. As $[\![x_j]\!]\,(\vec{\gamma}) = \gamma_j$ by assumption we get $\gamma_j\ \mathcal{R}_{\tau_j}\ M_j$.

Now the case of $\Gamma \vdash \lambda x.M : \sigma \to \tau$. Assume $\gamma_{n+1}\ \mathcal{R}_\sigma\ M_{n+1}$. By simplifying the goal both sides

$$[\![\lambda x.M]\!]\,(\vec{\gamma})(\gamma_{n+1}) = (\lambda x.\,[\![M]\!]\,(\vec{\gamma}, x))(\gamma_{n+1}) = [\![M]\!]\,(\vec{\gamma}, \gamma_{n+1})$$
$$(\lambda x.M)[\vec{M}/\vec{x}](M_{n+1}) = \lambda x.(M[\vec{M}/\vec{x}])(M_{n+1})$$

Since

$$\lambda x.(M[\vec{M}/\vec{x}])(M_{n+1}) \to^0 (M[\vec{M}/\vec{x}])[M_{n+1}/x]$$

By Lemma 3.27 the goal becomes

$$[\![M]\!]\,(\vec{\gamma}, \gamma_{n+1})\ \mathcal{R}_\tau\ M[\vec{M}/\vec{x}][M_{n+1}/x]$$

We apply the induction hypothesis on $\Gamma, x : \sigma \vdash M : \tau$ to the goal and we get what we wanted.

Now the case $\Gamma \vdash M_1 M_2 : \tau$. By induction hypothesis we have $[\![M_2]\!]\,(\vec{\gamma})\ \mathcal{R}_\tau\ (M_2)[\vec{M}/\vec{x}]$ which we can apply to the induction hypothesis that we get from M_1. So we get

$$[\![M_1]\!]\,(\vec{\gamma})\,[\![M_2]\!]\,(\vec{\gamma})\ \mathcal{R}_\tau\ (M_1[\vec{M}/\vec{x}])(M_2[\vec{M}/\vec{x}])$$

which is what we wanted to show.

Now the case for $\Gamma \vdash \texttt{fst}\ M : \tau_1$. By induction hypothesis

$$[\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1 \times \tau_2}\ M[\vec{M}/\vec{x}]$$

By definition of $\mathcal{R}_{\tau_1 \times \tau_2}$ we get

$$\pi_1([\![M]\!]\,(\vec{\gamma}))\ \mathcal{R}_{\tau_1}\ \texttt{fst}\ M[\vec{M}/\vec{x}]$$

Since $\pi_1([\![M]\!]\,(\vec{\gamma}))$ is by definition $[\![\texttt{fst}\ M]\!]\,(\vec{\gamma})$ and $\texttt{fst}\ (M[\vec{M}/\vec{x}])$ is equal to $(\texttt{fst}\ M)[\vec{M}/\vec{x}]$ we get what we wanted. The case for $\Gamma \vdash \texttt{snd}\ M : \tau_2$ is similar to the previous case.

The case $\Gamma \vdash \langle M, N\rangle : \tau_1 \times \tau_2$ is straightforward by definition of the logical relation, by Lemma 3.26 and by induction hypothesis.

Now for the case $\Gamma \vdash \texttt{unfold}\ M : \tau[\mu\alpha.\tau/\alpha]$ we want to show that

$$[\![\texttt{unfold}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ (\texttt{unfold}\ M)[\vec{N}/\vec{x}]$$

By induction hypothesis we know that

$$[\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\mu\alpha.\tau}\ (M[\vec{N}/\vec{x}])$$

which means that there exists M' and M'' such that

$$\texttt{unfold}\ (M[\vec{N}/\vec{x}]) \to^0_* M' \text{ and } M' \to^1 M''$$

and that $[\![M]\!]\,(\vec{\gamma}) \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$. By Lemma 3.28

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\vec{\gamma}))\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ M'$$

and since $\texttt{unfold}\ (M[\vec{N}/\vec{x}]) \to^0_* M'$ by Lemma 3.27 we get

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\vec{\gamma}))\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \texttt{unfold}\ (M[\vec{N}/\vec{x}])$$

By definition of the interpretation

$$[\![\texttt{unfold}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \texttt{unfold}\ (M[\vec{N}/\vec{x}])$$

which is what we wanted.

For the case $\Gamma \vdash \texttt{fold}\ M : \mu\alpha.\tau$ we want to show that

$$[\![\texttt{fold}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\mu\alpha.\tau}\ (\texttt{fold}\ M)[\vec{N}/\vec{x}]$$

First off, by definition of the substitution function $(\texttt{fold}\ M)[\vec{N}/\vec{x}])$ is equal to $\texttt{fold}\ (M[\vec{N}/\vec{x}])$. Thus, by definition of the logical relation we have to show that there exist M' and M'' such that

$$\texttt{unfold}\ (\texttt{fold}\ (M[\vec{N}/\vec{x}])) \to^0_* M'$$

$M' \to^1 M''$ and that $[\![\texttt{fold}\ M]\!]\,(\vec{\gamma}) \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$. Setting M'' to be $(M[\vec{N}/\vec{x}])$, we are left to show that

$$[\![\texttt{fold}\ M]\!]\,(\vec{\gamma}) \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M[\vec{N}/\vec{x}])$$

which is equal by definition of the interpretation function to

$$\mathsf{next}([\![M]\!]\,(\vec{\gamma})) \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}((M[\vec{N}/\vec{x}]))$$

The latter is equal by (3.27) to

$$\triangleright([\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ (M[\vec{N}/\vec{x}]))$$

which is true by the guarded recursive hypothesis.

For the case $\Gamma \vdash \texttt{inl}\ M : \tau_1 + \tau_2$ we have to prove that

$$[\![\texttt{inl}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2}\ \texttt{inl}\ M[\vec{M}/\vec{x}]$$

By definition of the interpretation function $[\![\texttt{inl}\ M]\!]\,(\vec{\gamma})$ is equal to

$$\eta(\mathsf{inl}([\![M]\!]\,(\vec{\gamma})))$$

By definition of the logical relation we have to prove that there exists M' such that $(\texttt{inl}\ M)[\vec{M}/\vec{x}] \Downarrow^0 \texttt{inl}\ M'$ and that $[\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1}\ M'$. The former is trivially true with $M' = M[\vec{M}/\vec{x}]$ and the latter is by induction hypothesis. The case for $\Gamma \vdash \texttt{inr}\ N : \tau_1 + \tau_2$ is similar to the previous case.

As for the case $\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \sigma$ we have to prove that

$$[\![\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma})\ \mathcal{R}_{\sigma}\ (\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}]$$

For this it suffices to prove

$$\begin{aligned}&[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2\to\sigma}\\&(\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}]\end{aligned}\tag{3.28}$$

and then applying this to $[\![L]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2}\ L[\vec{M}/\vec{x}]$. We prove (3.28) by guarded recursion thus assuming the statement is true later. Assume β of type $[\![\tau_1+\tau_2]\!]$, L of type $\mathtt{Term}_{\mathtt{FPC}}$ and $\beta\ \mathcal{R}_{\tau_1+\tau_2}\ L$. We proceed by case analysis on β which is of type $[\![\tau_1+\tau_2]\!]$ which is by definition $L([\![\tau_1+\tau_2]\!])$. In the case $\beta = \eta(\mathsf{inl}(\beta'))$, where β' is of type $[\![\tau_1]\!]$ we know by assumption that there exists L' s.t. $L \Downarrow^0 \mathtt{inl}\ (L')$ and $\beta'\ \mathcal{R}_{\tau_1}\ L'$. Since

$$[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma})(\eta(\mathsf{inl}(\beta'))) = [\![M]\!]\,(\vec{\gamma},\beta')$$

and

$$\mathtt{case}\ L\ \mathtt{of}\ x_1.M[\vec{M}\vec{x}]; x_2.N[\vec{M}/\vec{x}] \Rightarrow^0 M[\vec{M}/\vec{x}][L'/x_1]$$

by Lemma 3.27 we are left to prove

$$[\![M]\!]\,(\vec{\gamma},\gamma)\ \mathcal{R}_\sigma\ \ M[\vec{M}/\vec{x}][L'/x_1]$$

which is true by induction hypothesis.

The case $\beta = \eta(\mathsf{inr}(\beta'))$ where β' is of type $[\![\tau_2]\!]$ is similar to the previous one.

The interesting case is when $\beta = \theta_{\tau_1+\tau_2}(\beta')$, where β' is of type $\triangleright[\![\tau_1+\tau_2]\!]$. By induction hypothesis we know that $\theta_{\tau_1+\tau_2}(\beta')\ \mathcal{R}_{\tau_1+\tau_2}\ L$, thus there exist L' and L'' of type$\mathtt{Term}_{\mathtt{FPC}}$ such that $L \to^0_* L'$, $L' \to^1 L''$ and $\beta' \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(L'')$.

Recall that the hypothesis says that

$$\begin{aligned}&\triangleright([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma})\\&\qquad\mathcal{R}_{\tau_1+\tau_2\to\sigma}\\&(\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}])\end{aligned}$$

which is type equal to

$$\begin{aligned}&\mathsf{next}([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma}))\\&\qquad\triangleright\mathcal{R}_{\tau_1+\tau_2\to\sigma}\\&\mathsf{next}((\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}])\end{aligned}$$

By Lemma 3.24 we can apply the guarded hypothesis with

$$\beta' \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(L'')$$

thus getting

$$\begin{aligned}
&\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N \rrbracket\,(\vec{\gamma})) \circledast \beta' \\
&\qquad \triangleright\mathcal{R}_\sigma \\
&\mathsf{next}(((\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}])(L''))
\end{aligned}$$

Since $L' \rightarrow^1 L''$ we can apply Lemma 3.28 and obtain

$$\begin{aligned}
&\theta_\sigma(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N \rrbracket\,(\vec{\gamma})) \circledast \beta') \\
&\qquad \mathcal{R}_\sigma \\
&\mathtt{case}\ L'\ \mathtt{of}\ x_1.M[\vec{M}/\vec{x}]; x_2.N[\vec{M}/\vec{x}]
\end{aligned}$$

By Lemma 3.27 with the fact that $L \rightarrow^0_* L'$ we get

$$\begin{aligned}
&\theta_\sigma(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N \rrbracket\,(\vec{\gamma})) \circledast \beta') \\
&\qquad \mathcal{R}_\sigma \\
&\mathtt{case}\ L\ \mathtt{of}\ x_1.M[\vec{M}/\vec{x}]; x_2.N[\vec{M}/\vec{x}]
\end{aligned}$$

And finally by simplifying the left-hand side by using Lemma 3.19:

$$\begin{aligned}
&\theta_\sigma(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N \rrbracket\,(\vec{\gamma})) \circledast \beta') = \\
&\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N \rrbracket\,(\vec{\gamma})(\beta)
\end{aligned}$$

thus getting

$$\begin{aligned}
&\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N \rrbracket\,(\vec{\gamma})(\beta) \\
&\qquad \mathcal{R}_\sigma \\
&(\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}](L)
\end{aligned}$$

as we wanted. □

From the Fundamental lemma we can now prove computational adequacy.

Theorem 3.30 (Intensional Computational Adequacy)**.** *If $M : 1$ is a closed term then $M \Downarrow^k \langle\rangle$ iff $\llbracket M \rrbracket\,(*) = \delta^k(\eta(\star))$.*

Proof. The proof is similar to [PMB15]. □

From Theorem 3.30 one can deduce that whenever two terms have equal denotations they are contextually equivalent in a very intensional way, as we now describe. By a context, we mean a term $C[-]$ with a hole, and we say that $C[-]$ has type $\Gamma, \tau \rightarrow (-, 1)$ if $C[M]$ is a closed term of type 1, whenever $\Gamma \vdash M : \tau$.

Corollary 3.31. *Suppose $\Gamma \vdash M : \tau$ and $\llbracket M \rrbracket = \llbracket N \rrbracket$. If $C[-]$ has type $\Gamma, \tau \rightarrow (-, 1)$ and $C[M] \Downarrow^k \langle\rangle$ also $C[N] \Downarrow^k \langle\rangle$.*

3.6 Extensional Computational Adequacy

Our model of FPC is intensional in the sense that it distinguishes between computations computing the same value in a different number of steps. In this section we define a logical relation which relates elements of the model if they differ only by a finite number of computation steps. In particular, this also means relating $\bot$ to $\bot$.

To do this we need to consider global behaviour of computations, as opposed to the local (or finitely computable) behaviour captured by the guarded recursive lifting monad L. To understand what this means, consider the interpretation of $L1$ in the topos of trees as described in Section 3.2.1. For each number n, the set

$$L1(n) = \{\bot, 0, 1, \dots, n-1\}$$

describes computations terminating in at most $n-1$ steps or using at least n steps (corresponding to $\bot$). It cannot distinguish between termination in more than $n-1$ steps and real divergence. Our relation should relate a terminating value x in $L1(n)$ to any other terminating value, but not real divergence, which is impossible, if divergence cannot be distinguished from slow termination.

On the other hand, consider the partiality monad [Cap05] L^{gl} defined as the coinductive solution to the type equation

$$L^{\mathrm{gl}}A \cong A + L^{\mathrm{gl}}A \tag{3.29}$$

When interpreted in **Set**, $L^{\mathrm{gl}}1$ is $\bar{\omega}$, i.e., describes the set of all possible behaviors of a computation of unit type.

Coinductive types can be encoded in gDTT using guarded recursive types, following ideas of Atkey and McBride [AM13; Møg14]. The encoding uses universal quantification over clocks, which we now briefly recall, refering to [Biz+16] for details.

3.6.1 Universal quantification over clocks

In gDTT all types and terms are typed in a clock context, i.e., a finite set of names of clocks. For each clock κ, there is a type constructor $\triangleright$, a fixed point combinator, and so on. The development of this paper so far has been in a context of a single implicit clock κ.

If A is a type in a context where κ does not appear, one can form the type $\forall\kappa.A$, binding κ. This construction behaves in many ways similarly to polymorphic quantification over types in System F. There is an associated binding introduction form $\Lambda\kappa.(-)$ (applicable to terms, where κ does not appear free in the context), and elimination form $t[\kappa']$ having type $A[\kappa'/\kappa]$ whenever $t\colon \forall\kappa.A$.

The type system allows for a restricted elimination rule for $\triangleright$. If t is of type $\triangleright A$ in a context where κ does not appear free, then $\mathsf{prev}\,\kappa.t$ has type $\forall\kappa.A$. Using $\mathsf{prev}\,\kappa.$ we can define a term force:

$$\begin{aligned} &\mathsf{force} : \forall\kappa.\triangleright A \to \forall\kappa.A \\ &\mathsf{force} \stackrel{\text{def}}{=\!=} \lambda x.\,\mathsf{prev}\,\kappa.x[\kappa] \end{aligned} \tag{3.30}$$

The type constructor $\forall\kappa.(-)$ is modelled by taking sets of global elements. In particular, $\forall\kappa.L1$ is modelled as $\bar{\omega}$. In fact, one can prove in the type theory, that defining

$$L^{\mathrm{gl}}A \stackrel{\mathrm{def}}{=\!=} \forall\kappa.LA$$

gives a coinductive solution to (3.29), when κ is not free in A [Møg14].

For types A and B we say the two are type isomorphic if there exist two terms $f : A \to B$ and $g : B \to A$ such that $f(g(x)) \equiv x$ and $g(f(x)) = x$. When A is a type that does not mention any free clock variables and B is free with $x : A$ and a clock variable κ the following type isomorphism is derivable from the β and η rules of the constructors involved [Biz+16]

$$\forall\kappa.\Sigma(x : A).B \cong \Sigma(x : A).\forall\kappa.B \tag{3.31}$$

3.6.2 Global interpretation of types and terms

As said above, the model of FPC can be considered as being defined w.r.t. an implicit clock κ. To be consistent with the notation of the previous sections, κ will remain implicit in the denotations of types and terms, although one might choose to write e.g. $[\![\sigma]\!]^{\kappa}$ to make the clock explicit.

We define global interpretations of types and terms as follows:

$$[\![\sigma]\!]^{\mathrm{gl}} \stackrel{\mathrm{def}}{=\!=} \forall\kappa.\, [\![\sigma]\!]$$
$$[\![M]\!]^{\mathrm{gl}} \stackrel{\mathrm{def}}{=\!=} \Lambda\kappa.\, [\![M]\!]$$

such that if $\Gamma \vdash M : \tau$, then

$$[\![M]\!]^{\mathrm{gl}} \colon \forall\kappa.([\![\Gamma]\!] \to [\![\tau]\!])$$

Note that $[\![\sigma]\!]^{\mathrm{gl}}$ is a wellformed type, because $[\![\sigma]\!]$ is a wellformed type in context $\sigma\colon \mathtt{Type}_{\mathtt{FPC}}$ and $\mathtt{Type}_{\mathtt{FPC}}$ is an inductive type formed without reference to clocks or guarded recursion, thus κ does not appear in $\mathtt{Type}_{\mathtt{FPC}}$. By a similar argument $[\![M]\!]^{\mathrm{gl}}$ is welltyped.

Define for all σ the *delay* operator $\delta^{\mathrm{gl}}_{\sigma} \colon [\![\sigma]\!]^{\mathrm{gl}} \to [\![\sigma]\!]^{\mathrm{gl}}$ as follows

$$\delta^{\mathrm{gl}}_{\sigma}(x) \stackrel{\mathrm{def}}{=\!=} \Lambda\kappa.\delta_{\sigma}(x[\kappa]) \tag{3.32}$$

Similarly for LA, $\delta^{\mathrm{gl}}_{LA}(x) \stackrel{\mathrm{def}}{=\!=} \Lambda\kappa.\delta_{LA}(x[\kappa])$.

With these definitions we can lift the Adequacy Theorem to the global points.

Corollary 3.32 (Computational Adequacy). *If $M\colon 1$ is a closed term then for all n. $\forall\kappa.M \Downarrow^n \langle\rangle$ iff $\forall\kappa.\, [\![M]\!]\,(*) = \delta^n(\eta(\star))$.*

Proof. We first prove the left-to-right direction, so assume n and assume that for all κ, $M \Downarrow^n \langle\rangle$. By Adequacy Theorem 3.30 this implies $[\![M]\!]\,(*) = \delta^n(\eta(\star))$, thus proving the case. The other direction is similar by Adequacy Theorem. ☐

3.6.3 A weak bisimulation relation for the lifting monad

Before defining the logical relation on the interpretation of types, we define a relational version of the guarded recursive lifting monad L. If applied to the identity relation on a type A in which κ does not appear, we obtain a weak bisimulation relation similar to the one defined by Danielsson [Dan12] for the coinductive partiality monad.

Definition 3.33. *For a relation $R : A \times B \to \mathcal{U}$ define the lifting $LR\ : LA \times LB \to \mathcal{U}$ by guarded recursion and case analysis on the elements of LA and LB:*

$$\begin{aligned}
\eta(x)\ LR\ \eta(y) &\stackrel{\text{def}}{=\!=} x\ R\ y \\
\eta(x)\ LR\ \theta_{LB}(y) &\stackrel{\text{def}}{=\!=} \Sigma n, y'.\theta_{LB}(y) = \delta^n_{LB}(\eta(y'))\ \textit{and}\ x\ R\ y' \\
\theta_{LA}(x)\ LR\ \eta(y) &\stackrel{\text{def}}{=\!=} \Sigma n, x'.\theta_{LA}(x) = \delta^n_{LA}(\eta(x'))\ \textit{and}\ x'\ R\ y \\
\theta_{LA}(x)\ LR\ \theta_{LB}(y) &\stackrel{\text{def}}{=\!=} x \triangleright LR\ y
\end{aligned}$$

The lifting of R, intuitively, captures computations that differ for a finite amount of computational steps or both diverge. For example, $\bot$ as defined in Section 3.4 is always related to itself which can be shown by guarded recursion as follows. Suppose $\triangleright(\bot\ LR\ \bot)$. Since $\bot = \theta(\mathsf{next}(\bot))$, to prove $\bot\ LR\ \bot$, we must prove $\mathsf{next}(\bot) \triangleright LR\ \mathsf{next}(\bot)$. But, this type is equal to the assumption $\triangleright(\bot\ LR\ \bot)$ by (3.27).

We can also prove that LR is closed under addition of δ on either side.

Lemma 3.34. *If $R : A \times B \to \mathcal{U}$, and $x\ LR\ y$ then $x\ LR\ \delta_{LB}(y)$ and $\delta_{LA}(x)\ LR\ y$.*

Proof. Assume $x\ LR\ y$. We show $x\ LR\ \delta_{LB}(y)$. The proof is by guarded recursion, hence we first assume:

$$\triangleright(\Pi x : LA, y : LB.x\ LR\ y \Rightarrow x\ LR\ \delta_{LB}(y)). \tag{3.33}$$

We proceed by case analysis on x and y. If $x = \eta(x')$, then, since $x\ LR\ y$, there exist n and y' such that $y = \delta^n_{LB}(\eta(y'))$ and $x'\ R\ y'$. So then $\delta_{LB}(y) = \delta^{n+1}_{LB}(\eta(y'))$, from which it follows that $x\ LR\ \delta_{LB}(y)$.

For the case where $x = \theta_{LA}(x')$ and $y = \eta(v)$, it suffices to show that $\delta^n_{LA}(\eta(w))\ LR\ \eta(v)$ implies $\delta^n_{LA}(\eta(w))\ LR\ \delta_{LB}(\eta(v))$. The case of $n = 0$ was proved above. For $n = m + 1$ we know that if $\delta^n_{LA}(\eta(w))\ LR\ \eta(v)$ also $\delta^m_{LA}(\eta(w))\ LR\ \eta(v)$ holds by definition, and this implies

$$\triangleright(\delta^m_{LA}(\eta(w))\ LR\ \eta(v))$$

But this type can be rewritten as follows

$$\begin{aligned}
&\triangleright(\delta^m_{LA}(\eta(w))\ LR\ \eta(v)) \\
&\equiv \mathsf{next}(\delta^m_{LA}(\eta(w)) \triangleright LR\ \mathsf{next}(\eta(v))) \\
&\equiv \theta_{LA}(\mathsf{next}(\delta^m_{LA}(\eta(w))))\ LR\ \theta_{LB}(\mathsf{next}(\eta(v)))) \\
&\equiv \delta^n_{LA}(\eta(w))\ LR\ \delta_{LB}(\eta(v))
\end{aligned}$$

proving the case.

Finally, the case when $x = \theta_{LA}(x')$ and $y = \theta_{LB}(y')$. The assumption in this case is $x' \triangleright LR\ y'$, which means by (3.26),

$$\triangleright\left[x'' \leftarrow x', y'' \leftarrow y'\right].x''\ LR\ y''$$

By the guarded recursion hypothesis (3.33) we get

$$\triangleright\left[x'' \leftarrow x', y'' \leftarrow y'\right].x''\ LR\ \delta_{LB}(y'')$$

which can be rewritten to

$$\triangleright\left[x'' \leftarrow x', y'' \leftarrow y'\right].x''\ LR\ \theta_{LB}(\mathsf{next}(y'')) \tag{3.34}$$

By (3.25) there is an inhabitant of the type

$$\triangleright\left[x'' \leftarrow x', y'' \leftarrow y'\right].(\mathsf{next}(y'') = y')$$

and thus (3.34) implies $\triangleright[x'' \leftarrow x'].x''\ LR\ \theta_{LB}(y')$, which, by (3.27) and since $y = \theta_{LB}(y')$ equals $x' \triangleright LR\ \mathsf{next}(y)$. By definition, this is

$$\theta_{LA}(x')\ LR\ \theta_{LB}(\mathsf{next}(y))$$

which since $x = \theta_{LA}(x')$ is $x\ LR\ \delta_{LB}(y)$. □

We can lift this result to L^{gl} as follows. Suppose $R : A \times B \to \mathcal{U}$ and κ not in A or B. Define $L^{\mathrm{gl}}R : L^{\mathrm{gl}}A \times L^{\mathrm{gl}}B \to \mathcal{U}$ as

$$x\ L^{\mathrm{gl}}R\ y \stackrel{\mathrm{def}}{=\!=} \forall\kappa.x[\kappa]\ LR\ y[\kappa]$$

Lemma 3.35. *Let $x : L^{\mathrm{gl}}A$ and $y : L^{\mathrm{gl}}B$. If $x\ L^{\mathrm{gl}}R\ y$ then $x\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}(y)$ and $\delta^{\mathrm{gl}}(x)\ L^{\mathrm{gl}}R\ y$.*

Proof. Follows directly from Lemma 3.34. □

One might expect that $\delta_{LA}(x)\ LR\ \delta_{LB}(y)$ implies $x\ LR\ y$. This is not true, it only implies $\triangleright(x\ LR\ y)$. In the case of L^{gl}, however, we can use force to remove the $\triangleright$.

Lemma 3.36. *For all $x : L^{\mathrm{gl}}A$ and $y : L^{\mathrm{gl}}B$ and for all $R : A \times B \to \mathcal{U}$, if $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$ then $x\ L^{\mathrm{gl}}R\ y$.*

Proof. Assume $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$. We can rewrite this type by unfolding definitions and (3.27) as follows.

$$\begin{aligned}
\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y) &\equiv \forall\kappa.(\delta^{\mathrm{gl}}_{LA}(x))[\kappa]\ LR\ (\delta^{\mathrm{gl}}_{LB}(y))[\kappa] \\
&\equiv \forall\kappa.(\delta_{LA}(x[\kappa]))\ LR\ (\delta_{LB}(y[\kappa])) \\
&\equiv \forall\kappa.(\mathsf{next}(x[\kappa])\ \triangleright LR\ \mathsf{next}(y[\kappa])) \\
&\equiv \forall\kappa.\triangleright(x[\kappa]\ LR\ (y[\kappa]))
\end{aligned}$$

Using force this implies $\forall\kappa.(x[\kappa]\ LR\ (y[\kappa]))$ which is equal to $x\ L^{\mathrm{gl}}R\ y$. □

Lemma 3.37. *For all x of type $L^{\mathrm{gl}}A$ and y of type $L^{\mathrm{gl}}B$, if $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ y$ then $x\ L^{\mathrm{gl}}R\ y$.*

Proof. Assume $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ y$. Then by applying Lemma 3.35 we get $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$ and by applying Lemma 3.36 we get $x\ L^{\mathrm{gl}}R\ y$. □

$$\begin{aligned}
x \approx_1 y &\overset{\text{def}}{=\!=} x\ L(=_1)\ y \\
x \approx_{\tau_1+\tau_2} y &\overset{\text{def}}{=\!=} x\ L(\approx_{\tau_1} + \approx_{\tau_2})\ y \\
x \approx_{\tau_1\times\tau_2} y &\overset{\text{def}}{=\!=} \pi_1(x) \approx_{\tau_1} \pi_1(y) \text{ and } \pi_2(x) \approx_{\tau_2} \pi_2(y) \\
f \approx_{\sigma\to\tau} g &\overset{\text{def}}{=\!=} \Pi(x,y : [\![\sigma]\!]).x \approx_\sigma y \to f(x) \approx_\tau g(y) \\
x \approx_{\mu\alpha.\tau} y &\overset{\text{def}}{=\!=} x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} y
\end{aligned}$$

FIGURE 3.38.1: The logical relation $\approx_\tau$ is a predicate over denotations of τ of type $[\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$

3.6.4 Relating terms up to extensional equivalence

Figure 3.38.1 defines for each FPC type τ the logical relation $\approx_\tau : [\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$. The definition is by guarded recursion, and the well-definedness can be formalised using an argument similar to that used for well-definedness of θ in equation (3.13). The case of recursive types is well typed by Lemma 3.16. The figure uses the following lifting of relations to sum types.

Definition 3.38. *Let $R : A \times B \to \mathcal{U}$ and $R' : A' \times B' \to \mathcal{U}$. Define $(R + R') : (A + A') \times (B + B') \to \mathcal{U}$ by case analysis as follows (omitting false cases)*

$$\begin{aligned}
\mathit{inl}(x)\ (R + R')\ \mathit{inl}(y) &\overset{\text{def}}{=\!=} x\ R\ y \\
\mathit{inr}(x)\ (R + R')\ \mathit{inr}(y) &\overset{\text{def}}{=\!=} x\ R'\ y
\end{aligned}$$

The logical relation can be generalised to open terms and the global interpretation of terms as in the next two definitions.

Definition 3.39. *For $\Gamma \equiv x_1 : \sigma_1, \cdots, x_n : \sigma_n$ and for f, g of type $[\![\Gamma]\!] \to [\![\tau]\!]$ define*

$$f \approx_{\Gamma,\tau} g \overset{\text{def}}{=\!=} \Pi(\vec{x}, \vec{y} : [\![\vec{\sigma}]\!]).\vec{x} \approx_{\vec{\sigma}} \vec{y} \to f(\vec{x}) \approx_\tau g(\vec{y})$$

Definition 3.40. *For x, y of type $[\![\Gamma \to \tau]\!]^{\mathrm{gl}}$,*

$$x \approx^{\mathrm{gl}}_{\Gamma,\tau} y \overset{\text{def}}{=\!=} \forall\kappa.x[\kappa] \approx_{\Gamma,\tau} y[\kappa]$$

3.6.5 Properties of $\approx_{\Gamma,\tau}$

The weak-bisimulation relation behaves similarly to the applicative functor rule.

Lemma 3.41. *For all f, g of type $\triangleright [\![\tau \to \sigma]\!]$ and x, y of type $\triangleright [\![\tau]\!]$, if $f \triangleright \approx_{\tau\to\sigma} g$ and $x \triangleright \approx_\tau y$ then $(f \circledast x) \triangleright \approx_\sigma (g \circledast y)$.*

Proof. Assume $f \triangleright \approx_{\tau\to\sigma} g$ and $x \triangleright \approx_\tau y$. By Definition 3.26 $f \triangleright \approx_{\tau\to\sigma} g$ is $\triangleright [f' \leftarrow f, g' \leftarrow g].(f' \approx_{\tau\to\sigma} g')$ which by unfolding Figure 3.38.1 is

$$\triangleright \left[f' \leftarrow f, g' \leftarrow g\right].(\Pi(x, y : [\![\sigma]\!]).x \approx_\tau y \to f'(x) \approx_\sigma g'(y))$$

By applying x, y and $x \triangleright \approx_{\tau} \ y$ to the above we get

$$\triangleright \left[f' \leftarrow f, g' \leftarrow g, a \leftarrow x, b \leftarrow y\right].(f'(a) \ \approx_{\sigma} \ g'(b))$$

By (3.27)

$$\mathsf{next}\left[f' \leftarrow f, a \leftarrow x\right].(f'(a)) \triangleright \approx_{\sigma} \ \mathsf{next}\left[f' \leftarrow g, b \leftarrow y\right].(g'(b))$$

which by rule (3.23) is equal to

$$(f \circledast] x) \triangleright \approx_{\sigma} \ (g \circledast] y)$$

□

The following lemma generalises the second case of $\approx_1$ to all types.

Lemma 3.42. *Let x, y of type $\triangleright [\![\sigma]\!]$, if $(x \triangleright \approx_{\sigma} \ y)$ then $\theta_{\sigma}(x) \ \approx_{\sigma} \ \theta_{\sigma}(y)$*

Proof. We prove the statement first by guarded recursion. Thus, we assume the statement holds "later" and we proceed by induction on σ. All the cases for the types that are interpreted using the lifting – namely the unit type and the sum type – in Definition 3.33 hold by definition of the lifting relation.

Now the case for the function space. Assume $\sigma = \tau_1 \to \tau_2$ and assume x and y of type $\triangleright [\![\tau_1 \to \tau_2]\!]$ such that $x \triangleright \approx_{\tau_1 \to \tau_2} \ y$. We must show that if $x', y' : [\![\tau_1]\!]^{\kappa}$ and $x' \ \approx_{\tau_1} \ y'$ then $(\theta_{\tau_1 \to \tau_2}(x))(x') \ \approx_{\tau_2} \ (\theta_{\tau_1 \to \tau_2}(y))(y'))$.

So suppose $x' \ \approx_{\tau_1} \ y'$, then also $\triangleright(x' \ \approx_{\tau_1} \ y')$, which by (3.27) is equal to $\mathsf{next}(x') \triangleright \approx_{\tau_1} \ \mathsf{next}(y')$. By applying Lemma 3.41 to this and $x \triangleright \approx_{\tau_1 \to \tau_2} \ y$ we get

$$x \textcircled{\kappa} (\mathsf{next}\, x') \triangleright \approx_{\tau_2} \ y \textcircled{\kappa} \mathsf{next}\, y'$$

By induction hypothesis on τ_2, we get $\theta_{\tau_2}(x \textcircled{\kappa} (\mathsf{next}\, x')) \ \approx_{\tau_2} \ \theta_{\tau_2}(y \textcircled{\kappa} (\mathsf{next}\, y'))$.

We conclude by observing that by definition of θ, $\theta_{\tau_1 \to \tau_2}(x)(y) = \theta_{\tau_2}(x \textcircled{\kappa} \mathsf{next}(y))$, thus concluding the proof.

Now the case of the product. This is by definition of $\approx_{\tau_1 \times \tau_2}$ and by induction hypothesis.

Now the interesting case, namely the fixed-point. Assume $\phi \triangleright \approx_{\mu\alpha.\tau} \ \psi$. This is type equal to

$$\triangleright [x \leftarrow \phi, y \leftarrow \psi].(x \ \approx_{\mu\alpha.\tau} \ y)$$

By definition this is equal to

$$\triangleright [x \leftarrow \phi, y \leftarrow \psi].(x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \ y)$$

By guarded recursive hypothesis we get

$$\triangleright [x \leftarrow \phi, y \leftarrow \psi].(\theta_{\tau[\mu\alpha.\tau/\alpha]}(x) \ \approx_{\tau[\mu\alpha.\tau/\alpha]} \ \theta_{\tau[\mu\alpha.\tau/\alpha]}(y))$$

By (3.27) this is equal to

$$(\mathsf{next}\,[x \leftarrow \phi].(\theta_{\tau[\mu\alpha.\tau/\alpha]}(x))) \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \ (\mathsf{next}\,[y \leftarrow \psi].(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)))$$

This equals

$$(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \phi \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \ (\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \psi$$

By Lemma 3.42 $\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \phi$ is equal to $\theta_{\mu\alpha.\tau}(\phi)$ thus we can derive

$$\theta_{\mu\alpha.\tau}(\phi) \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \theta_{\mu\alpha.\tau}(\psi)$$

which by definition of $\approx_{\mu\alpha.\tau}$ is

$$\theta_{\mu\alpha.\tau}(\phi) \approx_{\mu\alpha.\tau} \theta_{\mu\alpha.\tau}(\psi)$$

$\square$

Lemma 3.43. *Let σ be a closed FPC type and let x and y of type $[\![\sigma]\!]$, if $x \approx_\sigma y$ then $\delta_\sigma(x) \approx_\sigma y$ and $x \approx_\sigma \delta_\sigma(y)$.*

Proof. The proof is by guarded recursion and then by induction on the type σ. Thus, assume this lemma holds "later". Now by induction on σ we first show the statement for the unit type. In this case $\approx_1$ is defined as the lifting of the equality $L(=)$ so Lemma 3.34 applies and similarly for the co-product case. For the case of the product assume $x \approx_{\tau_1\times\tau_2} y$ which by definition of the relation the left/right projection of x is related to the left/right projection of y. We show how to add a delay on the left-hand side. The other case is similar. We use the induction hypothesis on τ_1 and τ_2, thus getting $\delta_{\tau_1}(\pi_1(x)) \approx_{\tau_1} \pi_1(y)$ and the same for τ_2. Since $\delta_{\tau_1}(\pi_1(x))$ is equal by definition of δ for the product type to $\pi_1(\delta_{\tau_1\times\tau_2}(x))$ we get $\pi_1(\delta_{\tau_1\times\tau_2}(x)) \approx_{\tau_1} \pi_1(y)$ and same for τ_2. This means by definition that $\delta_{\tau_1\times\tau_2}(x) \approx_{\tau_1\times\tau_2} y$ thus concluding the case.

Now the case of the function space. Assume for two functions f and g that $f \approx_{\sigma\to\tau} g$ holds. For x and y such that $x \approx_\sigma y$, we have to prove $\delta_{\sigma\to\tau}(f)(x) \approx_\tau g(y)$. Assume thus such an x and y, from the hypothesis we know that $f(x) \approx_\tau g(y)$ holds. By induction hypothesis on τ we know that $\delta_\tau(f(x)) \approx_\tau g(y)$ and by definition of $\delta_{\sigma\to\tau}$ that is equal to $\delta_{\sigma\to\tau}(f)(x) \approx_\tau g(y)$ which is what we wanted.

Now the case for the recursive types. Assume $x \approx_{\mu\alpha.\tau} y$ which by definition is $x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} y$. By (3.26) we pull the later out together with two variables x' and y', thus getting $\triangleright [x' \leftarrow x, y' \leftarrow y] . x' \approx_{\tau[\mu\alpha.\tau/\alpha]} y'$. By the guarded recursion assumption we can add a delay, thus deriving $\triangleright [x' \leftarrow x, y' \leftarrow y] . x' \approx_{\tau[\mu\alpha.\tau/\alpha]} \delta_{\tau[\mu\alpha.\tau/\alpha]}(y')$. Note that the delay operator is the composition $\theta \circ \mathsf{next}$, thus y' appears under next. We can thus employ the eta rule of gDTT(3.25) and push y back in thus getting $\triangleright [x' \leftarrow x] . x' \approx_{\tau[\mu\alpha.\tau/\alpha]} \theta_{\tau[\mu\alpha.\tau/\alpha]}(y)$. By (3.26) we push the later back thus getting $x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y))$. By distributing the next over the application as in rule (3.2), $\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y))$ is equal to $\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \mathsf{next}(y)$ which is equal to $\theta_{\mu\alpha.\tau}(\mathsf{next}(y))$ by Figure 3.16.1. Finally, the logical relation on the case of the recursive type as in Figure 3.38.1 gives us $x \approx_{\mu\alpha.\tau} \theta_{\mu\alpha.\tau}(\mathsf{next}(y))$ thus proving the case and the lemma. Note that adding the delay to the other side of the relation is similar. $\square$

Lemma 3.44. *Let σ be a closed FPC type and let x, y of type $[\![\sigma]\!]^{\mathrm{gl}}$, if $x \approx^{\mathrm{gl}}_\sigma y$ then $x \approx^{\mathrm{gl}}_\sigma \delta^{\mathrm{gl}}_\sigma(y)$ and $\delta^{\mathrm{gl}}_\sigma(x) \approx^{\mathrm{gl}}_\sigma y$*

Proof. Direct from Lemma 3.43. $\square$

Contextual equivalence of FPC is defined in the standard way by observing convergence at unit type.

$$
\begin{aligned}
-[M] &\overset{\text{def}}{=\!=} M \\
(\lambda x.C)[N] &\overset{\text{def}}{=\!=} \lambda x.(C[N]) \\
(CM)[N] &\overset{\text{def}}{=\!=} (C[N])M \\
(MC)[N] &\overset{\text{def}}{=\!=} M(C[N]) \\
(\texttt{fst}\ C)[M] &\overset{\text{def}}{=\!=} \texttt{fst}\ C[M] \\
(\texttt{snd}\ C)[M] &\overset{\text{def}}{=\!=} \texttt{snd}\ C[M] \\
(\langle C, N\rangle)[M] &\overset{\text{def}}{=\!=} \langle C[M], N\rangle \\
(\langle N, C\rangle)[M] &\overset{\text{def}}{=\!=} \langle N, C[M]\rangle \\
(\texttt{fold}\ C)[N] &\overset{\text{def}}{=\!=} \texttt{fold}\ C[N] \\
(\texttt{unfold}\ C)[N] &\overset{\text{def}}{=\!=} \texttt{unfold}\ C[N] \\
(\texttt{inl}\ C)[M] &\overset{\text{def}}{=\!=} \texttt{inl}\ C[M] \\
(\texttt{inr}\ C)[M] &\overset{\text{def}}{=\!=} \texttt{inr}\ C[M] \\
(\texttt{case}\ C\ \texttt{of}\ x_1.M; x_2.N)[L] &\overset{\text{def}}{=\!=} \texttt{case}\ C[L]\ \texttt{of}\ x_1.M; x_2.N \\
(\texttt{case}\ L\ \texttt{of}\ x_1.C; x_2.N)[M] &\overset{\text{def}}{=\!=} \texttt{case}\ L\ \texttt{of}\ x_1.C[M]; x_2.N \\
(\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.C)[N] &\overset{\text{def}}{=\!=} \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.C[N]
\end{aligned}
$$

FIGURE 3.45.1: Definition of the “fill hole” function

Definition 3.45 (Contexts).

$$
\begin{aligned}
\mathsf{Ctx} := {} & - \mid \lambda x.\,\mathsf{Ctx} \mid \mathsf{Ctx}\,N \mid M\,\mathsf{Ctx} \\
& \mid \texttt{inl}\ \mathsf{Ctx} \mid \texttt{inr}\ \mathsf{Ctx} \mid \langle \mathsf{Ctx}, M\rangle \mid \langle M, \mathsf{Ctx}\rangle \mid \texttt{fst}\ \mathsf{Ctx} \mid \texttt{snd}\ \mathsf{Ctx} \\
& \mid \texttt{case}\ \mathsf{Ctx}\ \texttt{of}\ x_1.M; x_2.N \\
& \mid \texttt{case}\ L\ \texttt{of}\ x_1.\,\mathsf{Ctx}; x_2.N \mid \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.\,\mathsf{Ctx} \\
& \mid \texttt{unfold}\ \mathsf{Ctx} \mid \texttt{fold}\ \mathsf{Ctx}
\end{aligned}
$$

Intuitively, a context is a term that takes a term and returns a new term. Formally, we define a function that takes a syntactic context and turns it into a function that takes a term and returns a term. Thus, we define $\cdot[\cdot]$ as a function of type $\mathsf{Ctx} \times \texttt{Term}_{\texttt{FPC}} \to \texttt{Term}_{\texttt{FPC}}$ by induction on the context as in Figure 3.45.1.

We define the typing judgment for well-typed context as a function of type $\mathsf{Ctx} \times (\mathsf{Env} \times \texttt{Type}_{\texttt{FPC}})^2 \to \mathcal{U}$ as in Figure 3.45.2.

Definition 3.46. *Let $\Gamma \vdash M, N : \tau$. We say that M, N are contextually equivalent, written $M \approx_{CTX} N$, if for all contexts C of type $(\Gamma, \tau) \to (-, 1)$*

$$\forall \kappa. C[M] \Downarrow \langle\rangle \iff \forall \kappa. C[N] \Downarrow \langle\rangle$$

Finally we can state the main theorem of this section.

$$\frac{}{- : (\Gamma, \tau) \to (\Gamma, \tau)} \qquad \frac{C : (\Gamma, \tau) \to ((\Delta, x : \sigma'), \sigma)}{(\lambda x.C) : (\Gamma, \tau) \to (\Delta, \sigma' \to \sigma)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau' \to \sigma) \qquad \Delta \vdash N : \tau'}{CN : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{C : (\Gamma, \sigma) \to (\Delta, \tau') \qquad \Delta \vdash M : \tau' \to \sigma}{MC : (\Gamma, \sigma) \to (\Delta, \sigma)}$$

$$\frac{C : (\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)}{\texttt{unfold}\, C : (\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])} \qquad \frac{C : (\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])}{\texttt{fold}\, C : (\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1 \times \tau_2)}{\texttt{fst}\, C : (\Gamma, \tau) \to (\Delta, \tau_1)} \qquad \frac{C : (\Gamma, \tau) \to (\Delta, \tau_1 \times \tau_2)}{\texttt{snd}\, C : (\Gamma, \tau) \to (\Delta, \tau_2)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1) \qquad \Delta \vdash N : \tau_2}{\langle C, N \rangle : (\Gamma, \tau) \to (\Delta, \langle \tau_1, \tau_2 \rangle)} \frac{C : (\Gamma, \tau) \to (\Delta, \tau_2) \qquad \Delta \vdash M : \tau_1}{\langle M, C \rangle : (\Gamma, \tau) \to (\Delta, \langle \tau_1, \tau_2 \rangle)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2) \qquad \Delta, x_1 : \tau_1 \vdash M : \sigma \qquad \Delta, x_2 : \tau_2 \vdash N : \sigma}{\texttt{case}\ C\ \texttt{of}\ x_1.M; x_2.N : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{\Delta \vdash L : \tau_1 + \tau_2 \qquad C : (\Gamma, \tau) \to ((\Delta, x_1 : \tau_1), \sigma) \qquad \Delta, x_2 : \tau_2 \vdash N : \sigma}{\texttt{case}\ L\ \texttt{of}\ x_1.C; x_2.N : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{\Delta \vdash L : \tau_1 + \tau_2 \qquad \Delta, x_1 : \tau_1 \vdash M : \sigma \qquad C : (\Gamma, \tau) \to ((\Delta, x_2 : \tau_2), \sigma)}{\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.C : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1)}{\texttt{inl}\, C : (\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2)} \qquad \frac{C : (\Gamma, \tau) \to (\Delta, \tau_2)}{\texttt{inr}\, C : (\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2)}$$

FIGURE 3.45.2: Typing judgment for contexts

Theorem 3.47 (Extensional Computational Adequacy). *If* $\Gamma \vdash M, N : \tau$ *and* $\llbracket M \rrbracket^{\mathrm{gl}} \approx^{\mathrm{gl}}_{\Gamma,\tau} \llbracket N \rrbracket^{\mathrm{gl}}$ *then* $M \approx_{CTX} N$

We now give the proof of Theorem 3.47. The first lemma needed for the proof states that the interpretation of any term is related to itself. This needs to be proved by induction over terms, as the logical relation is not reflexive, as also noted by Escardó [Esc99]. As a counter example, consider a function $f\colon \llbracket 1 \rrbracket \to \llbracket 1 \rrbracket$ which diverges if its input takes a step and converges otherwise. Such a function is definable in the metalanguage, but not in FPC.

Lemma 3.48. *If* $\Gamma \vdash M : \sigma$, *then* $\llbracket M \rrbracket \approx_{\Gamma,\sigma} \llbracket M \rrbracket$.

Proof. Assume a variable context Γ such that $\Gamma \equiv x_1 : \sigma_1, \cdots, x_n : \sigma_n$. Assume also $\Gamma \vdash M : \sigma$ and $\vec{x}$, $\vec{y}$ of type $\llbracket \vec{\sigma} \rrbracket$ such that $\vec{x} \approx_{\vec{\sigma}} \vec{y}$. By Definition 3.39 we have to prove $\llbracket M \rrbracket(\vec{x}) \approx_{\sigma} \llbracket M \rrbracket(\vec{y})$. We proceed by induction on the typing judgment $\Gamma \vdash M : \sigma$. In the case of $\Gamma \vdash \langle\rangle : 1$ we have to prove that $\llbracket \langle\rangle \rrbracket(\vec{x}) \approx_1 \llbracket \langle\rangle \rrbracket(\vec{y})$. Since $\langle\rangle$ is interpreted as $\eta(v)$ and $\approx_1$ is defined to be the lifting of the equality, namely $L(=)$, the above is straigthforward. The introduction cases for the co-product are by definition and by induction hypothesis. In particular, when $\Gamma \vdash \texttt{inl}\ M : \tau_1 + \tau_2$ and $\Gamma \vdash \texttt{inr}\ M : \tau_1 + \tau_2$ by definition $\approx_{\tau_1+\tau_2}$ is $L(\approx_{\tau_1} + \approx_{\tau_1})$ which just requires to relate the denotations of M which are indeed related by induction hypothesis. Now the case for the elimination of the co-product. We have that $\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \tau$ and we have to prove that

$$\llbracket \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{x}) \approx_{\tau} \llbracket \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{y})$$

It suffices to prove that

$$\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{x}) \approx_{\sigma \to \tau} \llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{y}) \tag{3.35}$$

Thus that for all x, y s.t. $x \approx_{\tau_1+\tau_2} y$

$$\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{x})(x) \approx_{\tau} \llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{y})(y)$$

holds. We prove (3.35) by guarded recursion. Thus, we assume the statement holds "later" and we proceed by case analysis on x and y. When x is $\eta(x')$ and y is $\eta(y')$ the statement holds by definition and by induction hypothesis. Now we consider the case when x is $\theta_{\tau_1+\tau_2}(x')$ and y is $\eta(v)$. Since by assumption $x \approx_{\tau_1+\tau_2} y$ there exists n and w such that $x = \delta^n_{\tau_1+\tau_2}(\eta(w))$ and $w \approx_{\tau_1+\tau_2} v$. Because of this latter fact if w is $\mathsf{inl}(w')$ for some w' then also v is $\mathsf{inl}(v')$ for some v' and $w' \approx_{\tau_1} v'$. The other case of inr is similar. By inductive hypothesis we know that $\llbracket M \rrbracket(\vec{x}) \approx_{\tau_1 \to \tau} \llbracket M \rrbracket(\vec{y})$ and thus that $\llbracket M \rrbracket(\vec{x})(w') \approx_{\tau} \llbracket M \rrbracket(\vec{y})(v')$. Now we add n delays to the left-hand side by using Lemma 3.43 thus getting $\delta^n_{\tau}(\llbracket M \rrbracket(\vec{x})(w')) \approx_{\tau} \llbracket M \rrbracket(\vec{y})(v')$. By definition of the interpretation we know that $\llbracket M \rrbracket(\vec{x})(w')$ is equal to $\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket)(\vec{x})(\eta(w))$, thus $\delta^n_{\tau}(\llbracket M \rrbracket(\vec{x})(w'))$ is equal to $\delta^n_{\tau}(\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket)(\vec{x})(\eta(w)))$. By repeated application of Lemma 3.19 we push the delays on the argument of the left-hand side denotation, thus getting

$$\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\vec{x})(\delta^n_{\tau_1+\tau_2}(\eta(w)))$$

The relation thus becomes

$$\begin{aligned}&\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket\,(\vec{x})(\delta^n_{\tau_1+\tau_2}(\eta(w)))\ \approx_{\tau_1+\tau_2}\\ &\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x})(\eta(v))\end{aligned}$$

which is what we wanted to show. The last case is when x is $\theta_{\tau_1+\tau_2}(x')$ and y is $\theta_{\tau_1+\tau_2}(y')$. By guarded recursion we know that

$$\triangleright(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket\,(\vec{x})\ \approx_{\tau_1+\tau_2\to\tau}\ (\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y}))$$

By (3.27) we get

$$\begin{aligned}&\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x}))\triangleright\approx_{\tau_1+\tau_2\to\tau}\\ &\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket\,(\vec{y}))\end{aligned}$$

Since $\theta_{\tau_1+\tau_2}(x')\ \approx_{\tau_1+\tau_2}\ \theta_{\tau_1+\tau_2}(y')$, by definition of the logical relation we get $x' \triangleright\approx_{\tau_1+\tau_2}\ y'$. By Lemma 3.41 we apply this latter fact with the hypothesis which gives us

$$\begin{aligned}&\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x})\circledast x'\triangleright\approx_{\tau}\\ &\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y})\circledast y'\end{aligned}$$

By Lemma 3.42 we add a tick on both sides of the relation

$$\begin{aligned}&\theta_\tau(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x})\circledast x')\ \approx_{\tau}\\ &\theta_\tau(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y})\circledast y')\end{aligned}$$

By Lemma 3.19 we push the tick inside thus getting

$$\begin{aligned}&(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x})(\theta_{\tau_1+\tau_2}x'))\ \approx_{\tau}\\ &(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y})\theta_{\tau_1+\tau_2}(y'))\end{aligned}$$

and concluding the case.

We now show the case for the elimination of the product. We show the case for the first projection as the second projection is very similar. By induction hypothesis we know that $\llbracket M\rrbracket\,(\vec{x})\ \approx_{\tau_1\times\tau_2}\ \llbracket M\rrbracket\,(\vec{y})$ which by definition implies $\pi_1(\llbracket M\rrbracket\,(\vec{x}))\ \approx_{\tau_1}\ \pi_1(\llbracket M\rrbracket\,(\vec{y}))$ and same for τ_2. This concludes the case immediately by definition of the first projection. As for the introduction case of the product we know by induction hypothesis that $\llbracket M\rrbracket\,(\vec{x})\ \approx_{\tau_1}\ \llbracket M\rrbracket\,(\vec{y})$ and that $\llbracket N\rrbracket\,(\vec{x})\ \approx_{\tau_2}\ \llbracket N\rrbracket\,(\vec{y})$. By definition $\llbracket M\rrbracket\,(\vec{x})\ \approx_{\tau_1}\ \llbracket M\rrbracket\,(\vec{y})$ is equal to $\pi_1\,\llbracket\langle M,N\rangle\rrbracket\,(\vec{x})\ \approx_{\tau_1}\ \pi_1\,\llbracket\langle M,N\rangle\rrbracket\,(\vec{y})$ and the same for the second project. This proves by definition what we wanted. We now prove the case for function application. By induction hypothesis we know that $\llbracket M\rrbracket\ \approx_{\Gamma,\tau\to\sigma}\ \llbracket M\rrbracket$, hence we know that $\llbracket M\rrbracket\,(\vec{x})\ \approx_{\tau\to\sigma}\ \llbracket M\rrbracket\,(\vec{y})$. Also by induction hypothesis we know that $\llbracket N\rrbracket\ \approx_{\Gamma,\tau}\ \llbracket N\rrbracket$, hence we know that $\llbracket N\rrbracket\,(\vec{x})\ \approx_{\tau}\ \llbracket N\rrbracket\,(\vec{y})$. By composing the two we get

$$\llbracket M\rrbracket\,(\vec{x})\,\llbracket N\rrbracket\,(\vec{x})\ \approx_{\sigma}\ \llbracket M\rrbracket\,(\vec{y})\,\llbracket N\rrbracket\,(\vec{y})$$

which is equal to $\llbracket MN\rrbracket\,(\vec{x})\ \approx_{\sigma}\ \llbracket MN\rrbracket\,(\vec{y})$, which proves that $\llbracket MN\rrbracket\ \approx_{\Gamma,\sigma}\ \llbracket MN\rrbracket$.

As for the case for the lambda abstraction we have to prove that $[\![\lambda x.M]\!] \approx_{\Gamma,\tau\to\sigma} [\![\lambda x.M]\!]$. To prove this it suffices to show that

$$[\![\lambda x.M]\!]\,(\vec{x}) \approx_{\tau\to\sigma} [\![\lambda x.M]\!]\,(\vec{y})$$

for all related $\vec{x}$, $\vec{y}$ in the context $[\![\Gamma]\!]$. By definition of $\approx_{\tau\to\sigma}$ we can assume x and y of type $[\![\tau]\!]$ such that $x \approx_\tau y$ and prove $[\![M]\!]\,(\vec{x})(x) \approx_\sigma [\![M]\!]\,(\vec{y})(y)$. But this follows directly from the induction hypothesis, namely $[\![M]\!] \approx_{(\Gamma,x:\tau),\sigma} [\![M]\!]$ by applying the context arguments.

Now we prove the two cases for the recursive types. We first prove the case for `unfold` M of type $\tau[\mu\alpha.\tau/\alpha]$. We have to show that

$$[\![\texttt{unfold}\,M]\!](\vec{x}) \approx_{\tau[\mu\alpha.\tau/\alpha]} [\![\texttt{unfold}\,M]\!](\vec{y})$$

By induction hypothesis we know that $[\![M]\!](\vec{x}) \approx_{\mu\alpha.\tau} [\![M]\!](\vec{y})$ which by definition of $\approx_{\mu\alpha.\tau}$ is $[\![M]\!](\vec{x}) \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} [\![M]\!](\vec{y})$. By Lemma 3.42 we get

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!](\vec{x})) \approx_{\tau[\mu\alpha.\tau/\alpha]} \theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!](\vec{y}))$$

and by definition of the interpretation function this is what we wanted. Now the case for `fold` M of type $\mu\alpha.\tau$. By induction hypothesis we know that $[\![M]\!](\vec{x}) \approx_{\tau[\mu\alpha.\tau/\alpha]} [\![M]\!](\vec{y})$ by "nexting" the assumption we get $\triangleright([\![M]\!](\vec{x}) \approx_{\tau[\mu\alpha.\tau/\alpha]} [\![M]\!](\vec{y}))$ which is equal to $\mathsf{next}([\![M]\!](\vec{x}) \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \mathsf{next}([\![M]\!](\vec{y}))$. By definition of $\approx_{\mu\alpha.\tau}$ this is precisely $\mathsf{next}([\![M]\!](\vec{x}) \approx_{\mu\alpha.\tau} \mathsf{next}([\![M]\!](\vec{y}))$ which by definition of the interpretation function is

$$[\![\texttt{fold}\,M]\!](\vec{x}) \approx_{\mu\alpha.\tau} [\![\texttt{fold}\,M]\!](\vec{y})$$

□

Lemma 3.49. *For all terms M and N, if $[\![M]\!] \approx_{\Gamma,\tau} [\![N]\!]$ then for all contexts C such that $C : \Gamma,\tau \to \Delta,\sigma$, $[\![C[M]]\!] \approx_{\Delta,\sigma} [\![C[N]]\!]$*

Proof. Assume $[\![M]\!] \approx_{\Gamma,\tau} [\![N]\!]$ and $C : \Gamma,\tau \to \Delta,\sigma$. We proceed by induction on the typing judgement $C : \Gamma,\tau \to \Delta,\sigma$. Also, assume $\Delta \equiv x_1 : \sigma_1 .., x_n : \sigma_n$ and $\vec{x} \approx_{\vec{\sigma}} \vec{y}$. For the empty context the case follows directly from the assumption.

Now we prove the case for the application context on the right case. Assume a context of the form $M'C : (\Gamma,\tau) \to (\Delta,\sigma)$. We have to prove that $[\![M'C[M]]\!] \approx_{\Delta,\sigma} [\![M'C[N]]\!]$. By Lemma 3.48 we know that $[\![M']\!](\vec{x}) \approx_{\tau'\to\sigma} [\![M']\!](\vec{y})$. By definition of the typing context we know that C is of type $\Gamma,\tau \to \Delta,\tau'$ and that M' is such that $\Delta \vdash M' : \tau' \to \sigma$. Thus, by induction hypothesis we know that $[\![C[M]]\!](\vec{x}) \approx_{\tau'} [\![C[N]]\!](\vec{y})$ and by applying it to the hypothesis on M' we obtain $[\![M']\!](\vec{x})[\![C[M]]\!](\vec{x}) \approx_\sigma [\![M']\!](\vec{y})[\![C[N]]\!](\vec{y})$ which is equal to $[\![M'C[M]]\!](\vec{x}) \approx_\sigma [\![M'C[N]]\!](\vec{y})$ by applying the definition of the interpretation for the application.

Now the left case for function application context. Assume $CN' : (\Gamma,\tau) \to (\Delta,\sigma)$. We have to prove that $[\![C[M]N']\!](\vec{x}) \approx_\sigma [\![C[N]N']\!](\vec{y})$. By Lemma 3.48 we know that $[\![N']\!](\vec{x}) \approx_{\tau'} [\![N']\!](\vec{y})$. The context C has type $C : (\Gamma,\tau) \to (\Delta,\tau' \to \sigma)$, thus by induction hypothesis we have $[\![C[M]]\!](\vec{x}) \approx_{\tau'\to\sigma} [\![C[N]]\!](\vec{y})$. By applying the hypothesis on N' and the one on $C[M]$ we get $[\![C[M]]\!](\vec{x})[\![N']\!](\vec{x}) \approx_\sigma [\![C[N]]\!](\vec{y})[\![N']\!](\vec{y})$, which by definition of the interpretation we get what we wanted.

Now case for the introduction of the co-product. Assume a context $\texttt{inl}\ C$ of type $(\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2)$. By induction C has type $(\Gamma, \tau) \to (\Delta, \tau_1)$ so $[\![C[M]]\!](\vec{x}) \approx_{\tau_1} [\![C[N]]\!](\vec{y})$. By Lemma 3.48 and by applying the context parameters for Δ, we get $[\![\lambda x.\texttt{inl}\ x]\!](\vec{x}) \approx_{\Delta,\tau_1 \to \tau_1+\tau_2} [\![\lambda x.\texttt{inl}\ x]\!](\vec{y})$ by applying the hypothesis on $\lambda x.\texttt{inl}\ x$ and $C[M]$ conclude. The case for $\texttt{inr}\ C$ is similar to the previous one.

Now the case for the elimination of the co-product. For a context of the form $\texttt{case}\ L\ \texttt{of}\ x_1.C; x_2.N'$ of type $(\Gamma, \tau) \to (\Delta, \sigma)$ we have by induction that C has type $(\Gamma, \tau) \to ((\Delta, \tau_1,)\sigma)$ and thus by induction hypothesis we know by applying the context parameters that $[\![C[M]]\!](\vec{x}) \approx_{\tau_1,\sigma} [\![C[N]]\!](\vec{y})$. From this we also know that $[\![\lambda x_1.C[M]]\!](\vec{x}) \approx_{\tau_1 \to \sigma} [\![\lambda x_1.C[N]]\!](\vec{y})$ By Lemma 3.48 and by applying the context parameters we get $[\![\lambda x.\texttt{case}\ L\ \texttt{of}\ x_1.x(x_1); x_2.N']\!](\vec{x}) \approx_{(\tau_1 \to \sigma)}$ $[\![\lambda x.\texttt{case}\ L\ \texttt{of}\ x_1.x(x_1); x_2.N']\!](\vec{y})$. By applying the induction hypothesis to this latter fact we conclude. The case for $\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.C : (\Gamma, \tau) \to (\Delta, \sigma)$ and for The case for $\texttt{case}\ C\ \texttt{of}\ x_1.M; x_2.N : (\Gamma, \tau) \to (\Delta, \sigma)$ is similar to the previous one. Now the case for the lambda abstraction context. For a context of the form $\lambda x.C$ and type $(\Gamma, \tau) \to (\Delta, \sigma' \to \sigma)$ we have to prove that for x and y such that $x \approx_{\sigma'} y$, $[\![\lambda x.C[M]]\!](\vec{x})(x) \approx_{\sigma} [\![\lambda x.C[N]]\!](\vec{y})(y)$ holds. But this exactly the induction hypothesis instantiated with first $\vec{x}$ and $\vec{y}$, then with x and y.

Now the case for the terms of the recursive types. For a context $\texttt{unfold}\ C$ of type $(\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])$ we have by induction that C has type $(\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)$ and thus induction hypothesis we know that $[\![C[M]]\!](\vec{x}) \approx_{\mu\alpha.\tau}$ $[\![C[N]]\!](\vec{y})$. By Lemma 3.48 we know that $[\![\lambda x.\texttt{unfold}\ x]\!] \approx_{(\mu\alpha.\tau) \to (\tau[\mu\alpha.\tau/\alpha])}$ $[\![\lambda x.\texttt{unfold}\ x]\!]$. By applying this latter fact to the induction hypothesis we obtain $[\![\texttt{unfold}\ C[M]]\!](\vec{x}) \approx_{\tau[\mu\alpha.\tau/\alpha]} [\![\texttt{unfold}\ C[N]]\!](\vec{y})$ which is what we wanted. Now the case for the fold context, namely $\texttt{fold}\ C$ of type $(\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)$. By induction, C has type $(\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])$ and thus by induction hypothesis we know that $[\![C[M]]\!](\vec{x}) \approx_{\tau[\mu\alpha.\tau/\alpha]} [\![C[N]]\!](\vec{y})$. By Lemma 3.48 we get $[\![\lambda x.\texttt{fold}\ x]\!] \approx_{(\tau[\mu\alpha.\tau/\alpha]) \to (\mu\alpha.\tau)} [\![\lambda x.\texttt{fold}\ x]\!]$. By applying this latter fact to the induction hypothesis we get $[\![\texttt{fold}\ C[M]]\!](\vec{x}) \approx_{\mu\alpha.\tau}$ $[\![\texttt{fold}\ C[N]]\!](\vec{y})$ which is what we wanted. □

The global lifting of the logical relation is closed under context.

Lemma 3.50. *If* $\Gamma \vdash M, N : \tau$ *and* $[\![M]\!]^{\mathrm{gl}} \approx^{\mathrm{gl}}_{\Gamma,\tau} [\![N]\!]^{\mathrm{gl}}$ *then for all contexts* C *of type* $(\Gamma, \tau) \to (-, 1)$, $[\![C[M]]\!]^{\mathrm{gl}} \approx^{\mathrm{gl}}_{(-,1)} [\![C[N]]\!]^{\mathrm{gl}}$

Proof. By specialising Lemma 3.49. □

The following lemma states that if two computations of unit type are related then the first converges iff the second converges. Note that this lemma needs to be stated using the fact that the two computations are *globally related.*

Lemma 3.51. *For all* x, y *of type* $[\![1]\!]^{\mathrm{gl}}$, *if* $x \approx^{\mathrm{gl}}_{(-,1)} y$ *then*

$$\Sigma n.x = (\delta^{\mathrm{gl}}_1)^n(\eta(\star)) \Leftrightarrow \Sigma m.y = (\delta^{\mathrm{gl}}_1)^m(\eta(\star))$$

Proof. (Sketch). We show the left to right implication, so suppose $x = (\delta^{\mathrm{gl}}_1)^n(\eta(\star))$. The proof proceeds by induction on n. If $n = 0$ then since by assumption $\forall\kappa.x[\kappa] \approx_1 y[\kappa]$, by definition of $\approx_1$, for all κ, there exists an m

such that $y[\kappa] = \delta_1^m(\eta(\star))$. By type isomorphism (3.31), since m is a natural number, this implies there exists m such that for all κ, $y[\kappa] = \delta_1^m(\eta(\star))$ which implies $y = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$ by clock extensionality.

In the inductive case $n = n' + 1$, since by Lemma 3.37 $(\delta_1^{\mathrm{gl}})^{n'}([\![v]\!]^{\mathrm{gl}}) \approx_1^{\mathrm{gl}} y$, the induction hypothesis implies $\Sigma m.y = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$. ☐

Proof of Theorem 3.47. Suppose $[\![M]\!]^{\mathrm{gl}} \approx_{\Gamma,\tau}^{\mathrm{gl}} [\![N]\!]^{\mathrm{gl}}$ and that C has type $(\Gamma, \tau) \rightarrow (-, 1)$. We show that if $\forall\kappa.C[M] \Downarrow \langle\rangle$ also $\forall\kappa.C[N] \Downarrow \langle\rangle$. So suppose $\forall\kappa.C[M] \Downarrow \langle\rangle$. By definition this means $\forall\kappa.\Sigma n.C[M] \Downarrow^n \langle\rangle$. Since n is a natural number, i.e. a type that does not mention any clock variable, by type isomorphism (3.31) we have that there exists n such that $\forall\kappa.C[M] \Downarrow^n \langle\rangle$. By the Global Adequacy Corollary 3.32 we get $\forall\kappa.\ [\![C[M]]\!] = (\delta_1)^n(\eta(\star))$ which is equivalent to $[\![C[M]]\!]^{\mathrm{gl}} = (\delta_1^{\mathrm{gl}})^n(\eta(\star))$. We can apply Lemma 3.50 together with the assumption and get $[\![C[M]]\!]^{\mathrm{gl}} \approx_1^{\mathrm{gl}} [\![C[N]]\!]^{\mathrm{gl}}$, so by Lemma 3.51 there exists an m such that $[\![C[N]]\!]^{\mathrm{gl}} = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$ which means there exists an m $\forall\kappa.\ [\![C[N]]\!] = (\delta_1)^m(\eta(\star))$. By applying the Global Adequacy Corollary 3.32 once again we get $\forall\kappa.C[N] \Downarrow \langle\rangle$ as desired. ☐

3.7 Conclusions and Future Work

We have shown that programming languages with recursive types can be given sound and computationally adequate denotational semantics in guarded dependent type theory. The semantics is intensional, in the sense that it can distinguish between computations computing the same result in different number of steps, but we have shown how to capture extensional equivalence in the model by constructing a logical relation on the interpretation of types.

This work can be seen as a first step towards a formalisation of domain theory in type theory. Other, more direct formalisations have been carried out in Coq, e.g. [BKV09; Ben+10] but we believe that the synthetic viewpoint offers a more abstract and simpler presentation of the theory. Moreover, we hope that the success of guarded recursion for operational reasoning, mentioned in the introduction, can be carried over to denotational models of advanced programming language features in future work.

Future work also includes implementation of gDTT in a proof assistant, allowing for the theory of this paper to be machine verified. Currently, initial experiments are being carried out in this direction.

Acknowledgements

This research was supported by DFF-Research Project 1 Grant no. 4002-00442, from The Danish Council for Independent Research for the Natural Sciences (FNU).

Part III

Applications to step-indexing

Chapter 4

Verifying Exceptions in Low-level code with Separation Logic

Marco Paviotti and Jesper Bengtson

Abstract. Exceptions in low-level architectures are implemented as synchronous interrupts: upon the execution of a faulty instruction the processor jumps to a piece of code that handles the error. Previous work has shown that assembly programs can be written, verified and run using higher-order separation logic [JBK13]. However, execution of faulty instructions is then specified as either being undefined or terminating with an error. In this paper, we study synchronous interrupts and show their usefulness by implementing a memory allocator. This shows that it is indeed possible to write positive specifications of programs that fault. All of our results are mechanised in the interactive proof assistant Coq.

4.1 Introduction

Assembly code is difficult to prove correct. When verifying imperative programs, standard Hoare-logics often make implicit assumptions about the control flow of programs and assume that the code c in a triple $\{P\}c\{Q\}$ has one entry point and one exit point, even though it may internally contain loops and method calls. In assembly programs we cannot make this assumption as the control flows of these languages are inherently unstructured.

Control flow is altered primarily by two mechanisms – jump commands and interrupts. Jump commands allow developers to execute code stored nearly anywhere in memory; their use is an active choice, much like writing a loop or calling a method. Interrupts, on the other hand, occur either when something has gone catastrophically wrong (such as dividing by zero or reading from un-mapped memory) or when an action from the environment requires processing (such as the user pressing a key, a change to the file system is made, or the processor clock ticks).

While some of the aspects of interrupts might resemble that of function calls, there are substantial differences: synchronous interrupts are not called explicitly but triggers unpredictably as a result of a particular operation on a particular state, .e.g. division by zero, secondly, there cannot be infinitely many calls as after three nested interruptions the machine automatically reboots. Interrupts that trigger as a result of an error are typically referred to as synchronous, while asynchronous interrupts are external requests. Another name for synchronous interrupts is exceptions, due to their similarity with the exceptions encountered in languages like Java or ML, and we will use the terms interchangeably.

We build on the existing Coq [Tea12] formalisation of the x86 instruction set [Cor13] by Jensen et al. [Ken+13]. Their memory-model (explained in Section 4.3.1) is very close to that of the actual x86 chipset – control flow is implemented using jumps which are inherently unstructured and code is stored in memory. This allows for self-modifying code. Secondly, their program logic [JBK13] is able to handle non-structured control through jumps in a clean and concise manner (explained in Section 4.3.2).

In this paper we present a monadic semantics and a program logic to verify x86-assembly programs that feature synchronous interrupts. By building on this existing work we are able to model synchronous interrupts very closely to the way they run on real processors.

On x86-architectures, interrupts operate by using an Interrupt Description Table (IDT). The IDT is stored in memory and contains one pointer to a handler for every type of interrupt. When an interrupt is fired, the processors state is saved, the address to the interrupt handler is retrieved from the table and the code of the handler is executed. Barring catastrophic failure, the original processor state is then typically restored and its execution is resumed. Similarly to how Jensen stores code in memory, we store the IDT and the interrupt handlers in memory, which opens up for possibilities like having programs updating the various handlers dynamically.

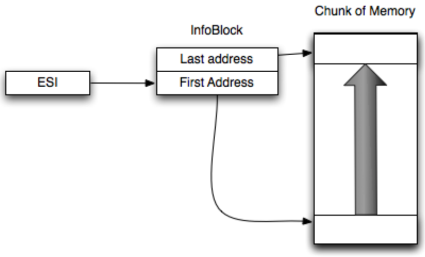
Our contributions are the following.

- We extend the semantics of Jensen et al. to support synchronous interrupts (Section 4.4.1).
- We add rules to the program logic to cover cases where synchronous interrupts are thrown, for example when reading from unmapped memory (Section 4.4.2), allowing users to verify programs that use interrupts.
- We verify a small memory allocator that uses synchronous interrupts (Section 4.5). To do this some technical – yet crucial – lemmas (Section 4.5.1) need to be shown for proving the specification of the allocator (Section 4.5.2)
- All of our results are mechanised in Coq.

The source code to our mechanisation can be found at `http://www.itu.dk/people/mpav/downloads/coqdev.zip`. The increment to the previous development amounts to 1084 lines of code. The code is compiled with `coqc` version `8.4pl3` with `OCaml 4.00.1`.

4.2 Memory allocation using exceptions

We use the standard AT&T syntax for assembly notation. For this example, '`mov` $r,\ v$' stores the value v in the register r, '$[r]$' dereferences a pointer stored in r, '`add` $r,\ v$' adds the value v to the value stored in the register r –



$$\begin{aligned}\mathsf{allocImp}(\mathit{info}, n, \mathit{fail}) \stackrel{\text{def}}{=}\ & \mathsf{mov\ ESI}, \mathit{info};\\ & \mathsf{mov\ EDI}, [\mathsf{ESI}];\\ & \mathsf{add\ EDI}, n;\\ & \mathsf{jc}\ \mathit{fail};\\ & \mathsf{cmp}\ [\mathsf{ESI}+4],\ \mathsf{EDI};\\ & \mathsf{jc}\ \mathit{fail};\\ & \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.\end{aligned}$$

FIGURE 4.0.1: Standard Allocation mechanism

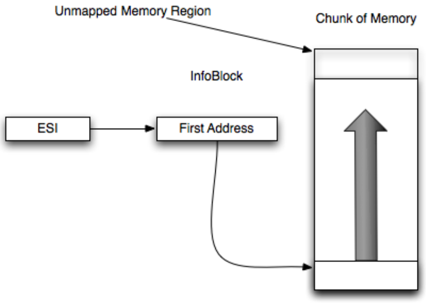
if the result to be stored in r exceeds the capacity of the register ($2^{32}-1$) the carry flag is set. The instruction 'jc a' jumps to address a if the carry flag is set. Finally, 'cmp r u' checks the carry flag. If the value stored in register r is greater than or equal to the value stored in u.

Jensen et al. [JBK13] implemented and verified a simple bound-and-check memory allocator, whose behavior and code is depicted in Figure 4.0.1. It takes three arguments, *info*, *n* and *fail*, where *info* is a pointer to an information block of two cells pointing to the beginning and at the end of the storage respectively, *n* is the number of words to be allocated, and *fail* is an address that the allocator jumps to if *n* bytes are not available. The program does two comparisons – the first checks if adding *n* to the memory start address causes the register value to wrap around (by resulting in a value greater than $2^{32}-1$), the second checks if that number is outside the memory available to the allocator. In both cases, the allocator jumps to *fail* if the test succeeds.

We verify an alternative version of this memory allocator using our new semantics and program logic for exceptions, whose behaviour and code is depicted in Figure 4.0.2. In our allocator we use the exception mechanism to jump to a handler in case of failure, thus there are no checks for overflow or memory bounds. Instead, we mark the end of the available memory with an unmapped location. Our allocator only has one argument *info* that is a single pointer to the start of memory. We only allow one word of memory to be allocated at a time and by writing a value to the word of memory we wish to allocate we will trigger an exception if that memory is unmapped, i.e. when the end of the memory available to the allocator has been reached. It is then up to the interrupt handler to catch the exception, but by jumping to the *fail* address it will mimic the behaviour of the handler in Figure 4.0.1.

4.2.1 Interrupt mechanism

Every interrupt is identified by a unique number which is an index into a record of pointers to the handlers. This list is commonly referred to as the *Interrupt Descriptor Table* (IDT) and it is a chunk of memory pointed to by the



$$
\begin{aligned}
\mathsf{allocImpExp}(\mathit{info}) \overset{\text{def}}{=\!=} \;& \mathsf{mov\ ESI}, \mathit{info};\\
& \mathsf{mov\ EDI}, [\mathsf{ESI}];\\
& \mathsf{mov}\ [\mathsf{EDI}],\ 0;\\
& \mathsf{add\ EDI}, 4;\\
& \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.
\end{aligned}
$$

FIGURE 4.0.2: Allocation mechanism with exceptions

IDT Register (IDTR) and further divided into records. Every record of the IDT contains a pointer to an interrupt handler.

In the Intel architecture when segmentation is used every address is uniquely calculated by giving a pair of addresses called *base* and *offset*. For example, the pair CS:EIP is the address of a piece of code with offset EIP inside the segment indicated by the value of CS. However, when referring to addresses inside the same segment, the segment register can be omitted.

When an interrupt triggers, the CPU looks up its number, saves the CS, EIP and the flags to the stack and jumps to the handler. In most operating system, only one segment is used so we chose not to worry about the segment selector – in particular the code selector – as it would be an easy fix if needed. The reader can safely skip this detail as we will not make any use in this paper.

Thus, the CPU is in charge of storing the return address in the stack and jump to the handler. Though, is responsibility of the programmer implementing a “safe” handler, i.e. a handler that leaves the machine in a state that the original program can continue from without faulting.

For example, a *transparent* interrupt handler is programmed such that there is no trace of its execution in the memory, i.e. the memory looks the same before and after the interrupt fired. This kind of handler saves the CPU state by pushing all the registers to the stack, handles the interrupt, and then restores the state as it was before it was interrupted. Finally, it executes the IRET instruction (Return from Interrupt). This signals the CPU to restore the triple CS:EIP and FLAGS, thus performing a far jump back where the program was interrupted.

However, the exception mechanism is slightly different in the presence of a faulty handler. If the handler produces an error the machine raises a *Double Fault* exception, which behaves the same as the other exceptions. Should this handler fail, the whole machine reboots. This situation is called *Triple Fault*.

4.3 Assembly semantics and logic

4.3.1 Semantics

In this section we present the semantics from Jensen et al. [JBK13]. The semantics of the assembly operates on a total state as all registers, flags, and memory :

$$\begin{aligned}\mathbb{S} \overset{\text{def}}{=\!=} &(\mathsf{reg} \to \mathsf{DWORD}) \times \\ &(\mathsf{flag} \to \{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}) \times \\ &(\mathsf{DWORD} \to (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\}))\end{aligned} \tag{4.1}$$

Here, $\mathbb{S}$ is made of three total functions: the register state, mapping each register to a DWORD (a 32-bit value); the flag state, mapping each flag to a boolean value or bottom; and the memory, mapping each 32-bit address to a BYTE (an 8-bit value), plus an unmapped value. The unmapped value stands for an inaccessible byte of memory. For example, it can be used when the memory is protected for some reason or inaccessible, e.g. some parts of the BIOS [Cor13]. Let E be the set of numbers from 0 to 255. These represent the errors that can occur, e.g. Division By Zero, General Protection and so on.

The result of a computation is either an error in E, an *unspecified behavior* or a result of type X along with the updated state:

$$R_X \overset{\text{def}}{=\!=} \mathsf{error}\ E \mid \mathsf{unspecified} \mid \mathsf{update}\ \mathbb{S}\ X$$

The semantics of the machine are monadic: programs are functions that takes a state $\mathbb{S}$ and produce a result. The type of a computation is the following:

$$ST\ X \overset{\text{def}}{=\!=} \mathbb{S} \to R_X$$

ST is a *state monad* with the usual *return* ($\eta : A \to ST\ A$) and *bind* operations »= : $ST\ A \to (A \to ST\ B) \to ST\ B$. We use let $x \leftarrow c;c'$ for c »= $\lambda x.c'$ and do $c;c'$ for c »= $\lambda\_.c'$. For each field of the state, we have read and write operations: $\mathsf{read}_{\mathsf{Flags}}$, $\mathsf{read}_{\mathsf{Reg}}$ and $\mathsf{read}_{\mathsf{Mem}}$ to read the value of flags, registers and memory locations respectively, and $\mathsf{set}_{\mathsf{Flags}}$, $\mathsf{set}_{\mathsf{Reg}}$, $\mathsf{set}_{\mathsf{Mem}}$ to write to the state.

The instruction set Instr is inductively defined. The interpreting function interprets an instruction $instr$ into an element $[\![intr]\!]$ of the monad $ST\ unit$, i.e. a function that takes a state a returns a result along with the modified state or an error. For example the interpretation for the jmp instruction:

$$[\![\mathsf{jmp}\ i]\!] \overset{\text{def}}{=\!=} \mathsf{set}_{\mathsf{Reg}}(\mathsf{EIP} := i)$$

A jump instruction is a computation that updates the EIP register with the address specified by the instruction.

The semantics of the machine are defined in terms of a step function of type *ST* unit which fetches and decodes an instruction and it inside the monad as follows:

$$\begin{aligned}\mathsf{step} \overset{\text{def}}{=\!=} & \ \mathsf{let}\ eip \leftarrow \mathsf{read}_{\mathsf{Reg}}(\mathsf{EIP});\\ & \ \mathsf{let}\ (instr, neweip) \leftarrow \mathsf{decode}(eip);\\ & \ \mathsf{do}\ \mathsf{set}_{\mathsf{Mem}}(\mathsf{EIP} := neweip); [\![instr]\!]\end{aligned}$$

where decode is a function which takes a 32-bit address and decodes it into an instruction and the new instruction pointer or returns an error. If the decoding fails, the entire step-function fails with the same error.

The semantics of the machine is defined by means of the function run of type $\mathbb{N} \times ST\ \mathsf{unit} \to ST\ \mathsf{unit}$ together with the step function defined above. run is defined recursively on its first argument and takes as input a natural number n which is the number of steps and a computation and executes that computation for n steps.

4.3.2 Separation Logic for low-level code

In this section we give a background on the program logic devised for reasoning about low-level code [JBK13].

A program logic or *specification logic* for a typical structured language has a judgment of the form $\vdash \{P\}c\{Q\}$, where P and Q are *assertion logic* formulas that represent the pre- and post-conditions of the command c respectively. Separation logic [RY04; Rey02; ORY01; IO01; COB03; COY07] is a logic for assertions. This logic is characterised by the *separating conjunction* $*$ where, for P and Q predicates on the heap, $P * Q$ means that P holds for one part of the heap and Q holds for another disjoint part of the heap.

However, in Hoare logic the statement $\vdash \{P\}c\{Q\}$ suggests that the program moves forward and if it terminates, eventually will reach a state satisfying the post-condition. In assembly language, however, it is not clear what the post-condition for a jump instruction would be. This instruction may jump to itself forever and never terminate. Nonetheless, the post-condition of the instruction would be satisfied after every each step, thus leading to a very intensional specification. Moreover, in assembly code is data. This means the code lives in the heap, hence it can be mentioned inside the assertion logic. This suggests that the specification logic should be able to model higher-order behaviours.

The solution is to define an unstructured continuation-passing-style specification logic where only preconditions are mentioned. A specification is a family of propositions indexed over natural numbers and assertions where the natural numbers is a step-index and the assertion plays the role of an *invariant extension* [BTY05]. In Coq this is formalised as:

$$\mathbb{N} \times (\Sigma \to \mathsf{Prop}) \to \mathsf{Prop}$$

where $\mathbb{N}$ is the set of natural numbers, Σ is the partial state obtained by taking the total state (4.1) and redefining it using partial functions instead of total ones and Prop is the proof-irrelevant universe of Coq. Every specification s of this type is downward closed on the index and upward closed w.r.t. the extension order on the separation logic formula. This will be explained shortly.

In the assertion logic, besides the separating conjunction, there are points-to relations for registers and flags, $\mapstochar\rhd$, and for the memory, $\mapsto$. As code is data, code can be specified in the assertion logic by using the $\mapsto$ relation, e.g. $i..j \mapsto c$ means the memory between i and j contains the program c. We are going to use the question mark $r?$, when r is a register, as syntactic sugar for $\exists v, r \mapstochar\rhd v$ and similarly for the memory. Separation logic entailments are solved conveniently within `Charge!` [Ben+11; BJB12], a comprehensive set of Coq tactics for working with a shallow embedding of a higher-order separation logic.

As an example, safe is the predicate which contains the set of pairs (k, R) such that the machine starting from a state satisfying R can run for k steps without faulting:

$$\mathsf{safe} = \lambda k.\lambda P.\ \forall \sigma \in P.\ \mathsf{run}\ \sigma\ k = \sigma' \text{ s.t. } \sigma' \neq \mathsf{error} \text{ and } \sigma' \neq \mathsf{unspecified} \quad (4.2)$$

Note that safe is indeed a predicate in the specification logic as for all k and P, if $(k, P) \in \mathsf{safe}$ holds then also $(k-1, P) \in \mathsf{safe}$ holds, and moreover, for all R, also $(k, P * R) \in \mathsf{safe}$ holds.

If S is a specification then $\vdash S$ means that for all k and R, $(k,\ R) \in S$ holds. For example, the statement $\vdash \mathsf{safe}$ means that the machine can run for any number of steps and from any state of the memory. This is obviously false, so we need a way to specify some sort of constraint on what the memory looks like.

For a specification S and a separation logic formula P, the *invariant extension* operator $\otimes$ gives a new specification $S \otimes P$. From the previous example, $\vdash \mathsf{safe} \otimes P$ means for all k and R, the machine starting from $P * R$ can run for k steps. The variant $S \oslash P$ is the read-only version for $\otimes$ and it is meant for disallowing self-modifying code. The $\otimes$ operator distributes over the implication and satisfies

$$(S \otimes P) \otimes Q = S \otimes (P * Q) \qquad (4.3)$$

among other rules. Moreover, the upward closure on the assertions implies the higher-order frame rule, $S \vdash S \otimes R$ [BTY05]. However, this fact is not used in the current paper.

In order to be able to specify a program behaviour that has a finite number of instructions we need to say what happens after the programs reaches its exit point (e.g. a jump out of the code fragment). Thus, a specification has the following continuation-passing style form:

$$\vdash (\mathsf{safe} \otimes \mathsf{EIP} \mapsto j * Q \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto i * P) \oslash i..j \mapsto c \qquad (4.4)$$

which states that a program c stored in the memory from the address i and the address j is safe to run from P *provided* that there is a continuation that runs safely from Q.

The $\otimes$ operator is an invariant extension operator in the following sense

$$(\mathsf{safe} \otimes P \Rightarrow \mathsf{safe} \otimes Q) \oslash i..j \mapsto c \otimes R \dashv\vdash \mathsf{safe} \otimes (P * R) \Rightarrow \mathsf{safe} \otimes (Q * R) \oslash i..j \mapsto c$$

where R is the assertion that is preserved through the computation. This can be shown by distributing R over the implication and by (4.3).

As an example, the mov instruction can be specified using the rule format 4.4 by instantiating as follows:

$$
\begin{aligned}
Q &\overset{\text{def}}{=} r_1 \mapsto\!\!\!\triangleright pd * pd \mapsto v_2 * r_2 \mapsto\!\!\!\triangleright v_2 \\
P &\overset{\text{def}}{=} r_1 \mapsto\!\!\!\triangleright pd * pd \mapsto v_1 * r_2 \mapsto\!\!\!\triangleright v_2 \\
c &\overset{\text{def}}{=} \mathsf{mov}\ [r_1], r_2
\end{aligned}
\tag{4.5}
$$

If the machine is safe to run from a state where EIP points to j and r_1 and r_2 are two registers containing a memory region that contains v_2 and the value v_2 itself respectively, then the machine is safe to execute the mov instruction which is located at the address i and the register r_1 is pointing to a different value.

Since the mov instruction does not alter the control flow the usual Hoare-triple specification can be used here: by setting $\{P\}\ c\ \{Q\}$ as notation for the specification in (4.4). Only in this specific case, i.e. when c is a code block that does not alter the control flow, the usual rules for composition and the frame rule applies.

This is not the case for the following example. An assembly program that sits in a tight loop forever is indeed safe:

$$
\vdash \mathsf{safe} \otimes \mathsf{EIP} \mapsto\!\!\!\triangleright i \oslash i..j \mapsto \mathsf{jmp}\ i \tag{4.6}
$$

In order to prove this statement, however, we need to prove the same statement after one step of computation, i.e. after the jump has been made. A convenient way to express this inside the logic is to use the $\rhd$ modality[1], pronounced "later". If S is a specification, $\rhd S$ means that S is true one step from now.

The $\rhd$ operator also introduces the notion of *Löb Induction* [App+07] which is made formal by the Löb rule

$$
\frac{\rhd S \vdash S}{\vdash S}\ \textsc{Löb}
$$

To prove that S holds it suffices to prove it assuming that "later" it will be true.

As an example, in order to prove (4.6) it suffices to prove the following:

$$
\rhd(\mathsf{safe} \otimes (\mathsf{EIP} \mapsto\!\!\!\triangleright i * i..j \mapsto \mathsf{jmp}\ i)) \vdash \mathsf{safe} \otimes (\mathsf{EIP} \mapsto\!\!\!\triangleright i * i..j \mapsto \mathsf{jmp}\ i)
$$

and the Löb rule allows us to do exactly that.

Note, the later-operator is not used to count the actual steps of the program as there is no tight connection between the modality and the number of steps the program performs. The $\rhd$ operator is only useful for breaking *circularity*, i.e. when a proof of a specification needs a proof of the same specification. Every program that jumps backwards and then after a while gets to the same point it was before is in this situation, e.g. (4.6).

Since we cannot know a priori whether the program is going to jump forwards or backwards every instruction that behaves as a jump needs to be specified with the $\rhd$ modality. For most of the instructions, on the other hand, this is not needed, as we just need a normal Hoare-triple form of specification where we assume that the next instruction is going to be safe.

---

[1] This modality was originally devised by Nakano [Nak00] and afterwards introduced by Birkedal et al. [Bir+12] as a synthetic form of step-indexing.

Finally, it is possible to compose specification of programs in a modular way. It is possible to give a more general composition rule where the program is just a generic separation logic formula. Though, we are only going to need a special case in Section 4.5 , we give an instantiation of the former, i.e. when the program is composed by an instruction and the rest of another program:

$$\frac{\begin{array}{l} \quad\vdash (\mathsf{safe} \otimes \mathsf{EIP} \mathrel{\mapstochar\triangleright} i_1 * Q_1 \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto i * P_1) \oslash i..i_1 \mapsto c_1 \\ P \vdash P_1 * R_{\mathrm{P}} \\ \quad\vdash (\mathsf{safe} \otimes Q \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mathrel{\mapstochar\triangleright} i_1 * Q_1 \otimes R_{\mathrm{P}}) \oslash i_1..j \mapsto c_1; c \end{array}}{\vdash (\mathsf{safe} \otimes Q \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mathrel{\mapstochar\triangleright} i * P) \oslash i..j \mapsto c_1; c} \tag{4.7}$$

The code snippet $c_1; c$ is the composition of the first instruction c_1 and the rest of the program c. In order to prove the whole program is safe to start from the beginning with memory in P we have to prove that P satisfies the precondition required by the first instruction c_1. Secondly, we have to prove the rest of the program is safe to execute from the part of the memory modified, namely Q_1 and part of P left untouched, namely R_P. Note that the code part of the specification is left unchanged. What changed is the program pointer to the memory where the code lies.

4.3.3 Memory representation

The formalisation covers a subset of the x86 allowing a user to write actual assembly x86 code, verify it, extract it as machine code and run it. To achieve this level of confidence the semantics of the machine has to be formalised in accordance with the specifications of the Intel machine [Cor13]. Values are encoded as vectors of booleans (lists of a set length), representing binary numbers.

```
Definition BITS (n: nat) := list n bool.
```

where n is the length of the list. Double words are defined similarly using the previous definition as a list of 32 bits:

```
Definition DWORD := BITS 32.
```

The definition of these types and their functions use modulo 2^n arithmetic in Coq. This is not suitable for points-to predicates in separation logic. Consider the predicate $p..q \mapsto v$ for p and q of type DWORD. Assuming this predicate it is not possible to infer that $p \leq q$ in arithmetic modulo 2^{32} as $p + 4$ might wrap around. A work-around consists in defining an additional type dependent on n that adds a `top` element representing the end of the memory:

```
Inductive Cursor (n: nat) := mkCursor (p: BITS n) | top.
```

a cursor is either a list of bits or the memory beyond the last representable address.

This representation of bit values pervades the whole framework and some challenges arise when trying to reason about points-to relations. In particular, when reasoning about lists of memory cells. We defer this issue to Section 4.5.1.

4.4 Semantics and logic for exceptions

4.4.1 Semantics for exceptions

We lift the semantics from Section 4.3.1 to catch the exceptions. The exception mechanism is implemented by the throw function. We currently do not handle double faults, but leave that behaviour unspecified.

```
$\mathsf{throw}(n : \texttt{nat}) \overset{\text{def}}{=\!=}$ let $level$ ← $\mathsf{read_{Reg}}$ INTL;
    if ($level$=0) then
        let $idt$ ← $\mathsf{read_{Reg}}$(IDTR);
        let $eip$ ← $\mathsf{read_{Reg}}$(EIP);
        do push($eip$);
        let $new$ ← $\mathsf{read_{Mem}}(idt + n * 4)$;
        $\mathsf{set_{Reg}}$(EIP := $new$)
        $\mathsf{set_{Reg}}$(INTL := $level + 1$)
    else
        raiseUnspecified
```

The former routine can be read as follows: if the INTL register is zero the machine raises it to one otherwise we leave the semantics unspecified; in the former case, the semantics looks up the address of the IDT, by reading the address from the IDTR, saves the address of the current execution point by pushing it to the stack pointed by ESP – denoted by push(eip) – fetches the address of the corresponding exception by looking up the value of the nth record inside the IDT, and sets this value (the address of the interrupt handler) to the EIP register.

In order to catch the error and throw an exception we define a catch function of type catch : *ST* unit → *ST* unit which takes a computation and gives a computation such that if the former ends up in a fault it throws an exception otherwise it returns the same result:

```
$\mathsf{catch}\ (c : ST\ \mathsf{unit})(s : \mathbb{S}) \overset{\text{def}}{=\!=}$ case $(c\ s)$ of
    | error($n$) ⇒ throw($n$)
    | $x \Rightarrow x$
end
```

Whenever we have a computation c, we obtain a computation catch c of type *ST* unit which turns errors into exceptions. We can then use the catch function with the interpretation function for instructions from Section 4.3.1: if $instr \in$ Instr then catch($[\![instr]\!]$) is the computation that throws an exception whenever the instruction $instr$ fails to execute.

Note that, we might have just as well modified each instruction semantics to throw the exception and remove the errors from the state. We chose instead to leave semantics untouched and build the catch function on top of it. We believe this is more modular and maintainable.

We substitute this term in the definition of step as follows:

$$\begin{aligned}\mathsf{step} \overset{\text{def}}{=\!=} & \ \mathsf{let}\ eip \leftarrow \mathsf{read}_{\mathsf{Reg}}(\mathsf{EIP});\\ & \ \mathsf{let}\ (instr, neweip) \leftarrow \mathsf{decode}(eip);\\ & \ \mathsf{do}\ \mathsf{set}_{\mathsf{Mem}}(\mathsf{EIP} := neweip); \mathsf{catch}([\![instr]\!])\end{aligned}$$

Note that the *only* difference between this step function and the one from the previous section is in the final command where we change $[\![instr]\!]$ to $\mathsf{catch}([\![instr]\!])$.

Whenever the instruction just fetched from the memory raises an exception, the machine jumps to the respective handler by updating the EIP register and the machine continues executing from there.

4.4.2 Specification logic for exceptions

In order to be able to reason about these exceptions we need assertions describing the state of the memory in which these events can be triggered. One of these is the read or write operation to an unmapped memory region. We define a predicate $l \mapsto !!$ as the set of states such that the location l maps to an unmapped location. This corresponds to a map that takes l to unmapped w.r.t the state definition (4.1).

Every instruction that tries to read or write from an unmapped locations now becomes a jump into the exception handler, provided that the IDT is present in memory.

The specification of the mov instruction of example (4.5) is turned into a jump-like specification using the later-operator. The rule has the following shape:

$$\vdash \triangleright(\mathsf{safe} \otimes Q \Rightarrow \mathsf{safe} \otimes P) \oslash i..j \mapsto \mathsf{mov}\ [r1], c \otimes \mathsf{Inv} \tag{4.8}$$

We are going to explain the meaning of P, Q and Inv soon. Informally, if the machine is "later" safe to run from inside the handler, where the interrupt level is set to 1 and the stack pointer points to the top of the stack where the return address j is stored, then it is safe to run from a state where the interrupt level is zero and the stack pointer points to a cell of memory such that the next cell is mapped.

First, we need an invariant stating the IDT is in the memory. With IDT[ExpGP/*fail*] we mean that the record in the IDT associated with the *general protection* exception contains the address *fail* which points to the handler. The location from l to $l+4$ is unmapped, while r_1 is pointing to l. Note that even though a complete IDT contains pointers to all handlers we do not need all of them, just the pointer to the handler associated with the general protection exception.

$$\mathsf{Inv} \overset{\text{def}}{=\!=} (r_1 \mapsto l * l..(l{+}4) \mapsto !! * \mathsf{IDT}[\mathsf{ExnGP}/\mathit{fail}]) \tag{4.9}$$

Secondly, the precondition in (4.10) says that the program is starting from i with interrupt level 0 and with the necessary space in the stack to allocate the return address:

$$P \overset{\text{def}}{=\!=} (\mathsf{EIP} \mapsto i * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto sp * ((sp-4)..sp)?) \tag{4.10}$$

Finally, we need a post-condition as in (4.11) stating that it is safe to jump inside the handler. Thus, at this point in time the program pointer will be the $fail$ address with interrupt level set to 1 and the return address on top of the stack:

$$Q \overset{\text{def}}{=\!=} \mathsf{EIP} \mapstochar\!\rightarrowtriangle \mathit{fail} * \mathsf{INTL} \mapstochar\!\rightarrowtriangle 1 * \mathsf{ESP} \mapstochar\!\rightarrowtriangle (sp - 4) * (sp - 4)..sp \mapsto j \tag{4.11}$$

Any instruction that makes use of an unmapped cell turns into a jump. In the Coq development we proved similar rules for the read version of mov and for push and pop instructions.

We proved that the push instruction is safe to execute when the parameter refers to an unmapped location:

$$\begin{aligned}
\vdash\quad & \forall i : \mathsf{DWORD}, j : \mathsf{DWORD}.\\
& (\triangleright(\mathsf{safe} \otimes (\mathsf{EIP} \mapstochar\!\rightarrowtriangle \mathit{fail} * \mathsf{INTL} \mapstochar\!\rightarrowtriangle 1 * \mathsf{ESP} \mapstochar\!\rightarrowtriangle (sp{-}4) * (sp{-}4)..sp \mapsto j)) \Rightarrow\\
& \mathsf{safe} \otimes (\mathsf{EIP} \mapstochar\!\rightarrowtriangle i * \mathsf{INTL} \mapstochar\!\rightarrowtriangle 0 * \mathsf{ESP} \mapstochar\!\rightarrowtriangle sp * (sp{-}4)..sp \mapsto v))\\
& \quad \oslash(i..j \mapsto \mathsf{push}[r])\\
& \quad \otimes(r \mapstochar\!\rightarrowtriangle l * l..(l{+}4) \mapsto\ !! * \mathsf{IDT}[\mathsf{ExnGP}/\mathit{fail}])
\end{aligned} \tag{4.12}$$

the structure is the same as in rule (4.8), but here we *read* from an unmapped memory address and the destination is the top of the stack. It may be worth noting that if the destination address is unmapped, i.e. the top of the stack, this rule is not sound: the push instruction is going to fail after having written a value on the stack which is unmapped and the interrupt mechanism is also going to store the return address on the stack resulting in a double fault which for the same reason leads to a triple fault.

The pop rule, similarly, reads from the stack and put the value on the cell pointed by the register r:

$$\begin{aligned}
\vdash\quad & \forall ij : \mathsf{DWORD}.\ (\triangleright(\mathsf{safe} \otimes (\mathsf{EIP} \mapstochar\!\rightarrowtriangle \mathit{fail} * \mathsf{INTL} \mapstochar\!\rightarrowtriangle 1 * (sp - 4)..sp \mapsto j *\\
& \mathsf{ESP} \mapstochar\!\rightarrowtriangle (sp - 4)))\\
& \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapstochar\!\rightarrowtriangle i * \mathsf{INTL} \mapstochar\!\rightarrowtriangle 0 * (sp - 4)..sp \mapsto spv * \mathsf{ESP} \mapstochar\!\rightarrowtriangle sp))\\
& \quad \oslash(i..j \mapsto (\mathsf{pop}[r]))\\
& \quad \otimes\mathsf{IDT}[\mathsf{ExnGP}/\mathit{fail}] * r \mapstochar\!\rightarrowtriangle l * l..(l + 4) \mapsto\ !!
\end{aligned} \tag{4.13}$$

the structure of the rule is again similarly to the previous ones and again, the rule would be unsound if we tried to pop a value from a stack whose memory is unmapped.

4.5 Memory allocation using exceptions

In this section we show how to prove the allocator correct. We first show how to reason about predicates that talk about chunks of memory with boundaries. More specifically, we need to be able to decide whether the current cell in a storage is available or not. Secondly, we use this result to prove the allocator correct.

4.5.1 The memory datatypes in Coq

As explained in Section 4.3.3 32-bit addresses binary are represented by the DWORD type. On the other hand, the points-to relation is a function from

Cursor $32 \times$ Cursor 32 to an assertion logic formula such that for all p, q and v, $p..q \mapsto v \vdash p \leq q$. The type Cursor 32 is making sure that p is actually smaller that q and that no wrap-around has occurred.

On the other hand, when using this predicate we need to know that the cursor represents a valid location in the memory and not the top element in order to reason about 32-bit address. Since a DWORD is a Cursor we can state $p..q \mapsto v$ for p and q of type DWORD.

Although, for convenience many instruction rules use $p \mapsto v$ as a notation for $\exists(q : \mathsf{Cursor}\ 32), p..q \mapsto v$. This implies that if we had more information about q by using these rules we would lose it. We worked around this problem by restating and proving the instruction rules, but in other cases it turned out to be easier to prove the following lemma:

Lemma 4.1. *Let p, q be two* DWORD*s and v a value also of type* DWORD*, $p..q \mapsto v \vdash q = p + 4$*

which required some non-trivial Coq hacking. The reader should note the subtlety of this lemma. The predicates p and q are cursors that came from a DWORD. Hence their value cannot be `top`. This means that the predicate $p..q \mapsto v$ is implicitly saying that $p + 4$ is not wrapping around since p and q are cursors and in this datatype the addresses are sequential.

By virtue of this effort we can now decide whether a chunk of memory is at its end or not:

Lemma 4.2. *Let base and limit be of type* Cursor *and let buf be a list of memory cells of type* DWORD*. If $\mathit{base}..\mathit{limit} \mapsto \mathit{buf} * \mathit{limit}..(\mathit{limit} + 4) \mapsto\ !!$ then either*

$$\exists s_1.\mathit{base}..(\mathit{base} + 4) \mapsto s_1 * \exists s_2.(\mathit{base} + 4)..\mathit{limit} \mapsto s_2 * \mathit{limit}..(\mathit{limit} + 4) \mapsto\ !! \tag{4.14}$$

or

$$\mathit{base} = \mathit{limit} \tag{4.15}$$

Proof. The proof is by case analysis on buf. If buf is the empty list then this implies base is equal to limit, thus satisfying (4.15).

If buf is composed by a cell a and the rest of the list l then there exists a p of type Cursor such that $\mathit{base}..p \mapsto a$ and $p..\mathit{limit} \mapsto l$. We proceed by case analysis on p. If p is a DWORD by Lemma 4.1 then p is equal $\mathit{base} + 4$, thus satisfying (4.14). If p is $\top$ then $\mathit{base}..\top \mapsto a$ is false. Since this was an assumption the case is vacuously true. □

4.5.2 Specification for the allocator

The specification for the example in Figure 4.0.2 has the following pattern,

$$\mathsf{allocSpec} \stackrel{\mathrm{def}}{=} \vdash ((\mathsf{safe} \otimes Q_1 \wedge \mathsf{safe} \otimes Q_2) \Rightarrow \mathsf{safe} \otimes P) \oslash i..j \mapsto c \otimes \mathsf{Inv}$$

Two continuations, namely Q_1 and Q_2 are defined to state what happens upon success and failure, a pre-condition P together with an invariant Inv specifying that there exists a storage whose ends are bounded by an unmapped memory region and that there exists an IDT containing the pointers to the handlers.

More precisely, the precondition is defined as follows:

$$P \overset{\text{def}}{=} \mathsf{EIP} \mapsto i * \mathsf{INTL} \mapsto 0 * \mathsf{EDI}? * \mathsf{ESP} \mapsto sp * (sp{-}4..sp)? \tag{4.16}$$

The EIP points to the beginning of the code, INTL is the register keeping track of the level of interruptions, EDI is a temporary register and ESP is the stack pointer.

We have to ensure there is a handler taking care that machine will be safe upon failure. This is stated by the first of the two post-conditions:

$$Q_1 \overset{\text{def}}{=} \mathsf{EIP} \mapsto fail * \mathsf{INTL} \mapsto 1 * \mathsf{EDI}? * \mathsf{ESP} \mapsto (sp{-}4) * (sp{-}4..sp)? \tag{4.17}$$

which states that there exists a handler which is going to take on the computation from the address $fail$ with the INTL set to 1 and the stack pointer containing the return address to the original code.

Also, we make sure the machine will be safe upon success. We do this by defining the other post-condition:

$$\begin{aligned} Q_2 \overset{\text{def}}{=} & \mathsf{EIP} \mapsto j * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto sp * ((sp{-}4)..sp)?* \\ & \exists p, \mathsf{EDI} \mapsto (p{+}4) * (p..(p+4))? \end{aligned} \tag{4.18}$$

which states that there is a program which is safe run from the address j with the EDI register pointing to the end of the allocated memory and with the interrupt level set at zero in case the allocator succeeds.

Furthermore, we have the following invariant:

$$\begin{aligned} \mathsf{Inv} \overset{\text{def}}{=} & \exists base\ count. infoBlock \mapsto base * \exists s, base..count \mapsto s \\ & * count..(count+4) \mapsto !! * \mathsf{IDTR} \mapsto idt \\ & * \mathsf{Flags} * \mathsf{IDT}[exn.ExnGP/fail] \end{aligned} \tag{4.19}$$

which states that the information block *infoBlock* points to a chunk of memory delimited on top by the unmapped region and the Interrupt Descriptor Table is properly set up in the memory.

4.5.3 Proof of the specification

We prove that implementation of the allocator respects the specification:

Theorem 4.3. *Let P, Q_1, Q_2 and* Inv *as respectively in (4.16), (4.17), (4.18) and (4.19). Moreover, let c be the code in Figure 4.0.2. The specification*

$$\vdash ((\mathsf{safe} \otimes Q_1 \land \mathsf{safe} \otimes Q_2) \Rightarrow \mathsf{safe} \otimes P) \oslash i..j \mapsto c \otimes \mathsf{Inv}$$

is sound. `[Coq proof]`

Proof. (Sketch). By unfolding the definition of the program there exists i_1, i_2,i_3 and i_4 of type DWORD pointing at each single instruction of the program:

$$\begin{aligned} i..i_1 &\mapsto \mathsf{mov\ ESI}, \mathit{infoBlock}\ * \\ i_1..i_2 &\mapsto \mathsf{mov\ EDI}, [\mathsf{ESI}]\ * \\ i_2..i_3 &\mapsto \mathsf{mov}\ [\mathsf{EDI}], 0\ * \\ i_3..i_4 &\mapsto \mathsf{add\ EDI}, 4\ * \\ i_4..j &\mapsto \mathsf{mov}\ [\mathsf{ESI}], \mathsf{EDI} \end{aligned}$$

Proving the first two instructions correct is only a matter of applying the proper rules using the composition rule (4.7).

First, we apply the instruction rule for the first instruction by instantiating it with the proper parameters:

$$\begin{aligned} &\vdash \mathsf{safe} \otimes \mathsf{EIP} \mapsto i_1 * \mathsf{ESI} \mapsto \mathit{infoBlock} \Rightarrow \mathsf{safe} \otimes \mathsf{EIP} \mapsto i * \mathsf{ESI}? \\ &\quad \oslash\, i..i_1 \mapsto \mathsf{mov\ ESI}, \mathit{infoBlock} \end{aligned}$$

For the second instruction we apply the following rule:

$$\begin{aligned} &\vdash \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i_2 * \mathsf{EDI} \mapsto v * \mathsf{ESI} \mapsto \mathit{infoBlock} * \mathit{infoBlock} \mapsto \mathit{base}) \\ &\quad \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i_1 * \mathsf{EDI}? * \mathsf{ESI} \mapsto \mathit{infoBlock} * \mathit{infoBlock} \mapsto \mathit{base}) \\ &\quad \oslash\, i_1..i_2 \mapsto \mathsf{mov\ EDI}, [\mathsf{ESI}] \end{aligned}$$

We end up with the following precondition which we name P' with the program pointer pointing to the third instruction:

$$\begin{aligned} P' = {} &\mathsf{safe} \otimes \mathsf{EIP} \mapsto i_2 * \mathsf{EDI} \mapsto \mathit{base} * \mathsf{ESI} \mapsto \mathit{infoBlock} * \mathit{infoBlock} \mapsto \mathit{base} \\ &* \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto \mathit{sp} * (\mathit{sp} - 4)..\mathit{sp} \mapsto \mathit{spval} * \mathsf{Inv} \end{aligned}$$

The instruction in i_2 is going to perform a write operation to the location pointed by base. By unfolding the invariant Inv we know there exists base and limit of type DWORD bounding the memory:

$$\begin{aligned} &\mathit{infoBlock} \mapsto \mathit{base} * \mathit{base}..\mathit{limit} \mapsto s * \mathit{limit}..(\mathit{limit} + 4) \mapsto !! * \mathsf{IDTR} \mapsto \mathit{idt} * \\ &\mathsf{Flags} * \mathsf{IDT}[\mathsf{ExnGP}/\mathit{fail}] \end{aligned}$$

on the other hand we do not know whether there is space left between them. So we case analyse the memory chunk by applying Lemma 4.2 thus getting

$$\begin{aligned} &\mathit{base}..(\mathit{base} + 4) \mapsto s * (\mathit{base} + 4)..\mathit{limit} \mapsto s * \mathit{limit}..(\mathit{limit} + 4) \mapsto !! \\ &\vee\ \mathit{base} = \mathit{limit} * \mathit{base}..\mathit{base} + 4 \mapsto\ !! \end{aligned}$$

This assertion is indeed part of P'. Thus, when applying (4.7) we will have to prove that P' implies the precondition of the instruction that we are going to use. This means the disjunction above will appear in the negative position. So we have to first split the disjunction into two sub cases. For the case in which the memory has run out we will apply (4.8) and for the other we will apply (4.5). We skip the second case as it is the standard one, i.e. the memory is available and the program goes through successfully satisfying post-condition Q_2.

In the case where $\mathit{base} = \mathit{limit}$ the move operation performs a write operation into the unmapped location. Let P'' be the precondition obtained from P' where $base = limit$. We use rule (4.8) instantiated as follows:

$$\begin{aligned}
&\triangleright(\mathsf{safe}\otimes(\mathsf{EIP} \mapsto fail * \mathsf{INTL} \mapsto 1 * \mathsf{ESP} \mapsto sp - 4 * (sp - 4)..sp \mapsto j)) \\
&\qquad \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i2 * \mathsf{INTL} \mapsto 0 * \mathsf{ESP} \mapsto sp * (sp - 4)..sp \mapsto spval) \\
&\qquad \oslash (i2..i3 \mapsto (\mathsf{mov}[\mathsf{EDI}], 0)) \\
&\qquad \otimes (\mathsf{EDI} \mapsto limit * limit..(limit + 4) \mapsto \,!! * \mathsf{IDTR} \mapsto \mathsf{IDT}[\mathsf{ExnGP}/fail])
\end{aligned}$$

This rule can be turned into a pattern suitable for applying (4.7) by commuting $\otimes$ with $\oslash$ and distributing $\otimes$ over the implication. We have to prove that P'' entails the precondition of the instruction rule above. In solving this entailment all the resource get consumed and the post-condition Q implies the post-condition for the instruction in i_2 □

4.6 Related Work

The work most closely related to ours is naturally the work by Jensen et al. that we build off of [JBK13]. It allows specifications to be written in a clean and intuitive manner even for code that does not follow a basic-block like structure with only one entry and one exit point. It should be noted, however, that there are program logics that use Hoare triples on code with multiple exit points, such as programs containing break-statements from loops. One example is Appel's mechanised semantics library in Coq for C-minor [App+14] and the mechanisation of x86 and ARM assembly by Myreen et al. in HOL4 [MG07]. Both these mechanisations have special post-conditions that are used to handle non-structured control flow. To our mind, the higher-order frame connective ($\otimes$) is a better solution as it provides a uniform way to write specifications for programs regardless of their internal control flow. Other relevant work includes Chlipala's Bedrock framework that allows assembly-like programs to be verified in Coq [Chl11]. Chlipala's work focusses heavily on automation and the expressivity of the specification logic is toned down somewhat. None of the work mentioned above currently support interrupts.

Seminal work on mechanising interrupts was made by Feng et al. in 2008 [Fen+08]. They focus on what they call an Abstract Interrupt Machine, which is an abstract model of a machine supporting both synchronous and asynchronous interrupts. They do, however, make some simplifications. Scheduling, for instance, is handled using a scheduling queue that is part of the model rather than being coded using interrupts from the clock. In our approach, the interrupt handlers are programs, just like any other, allowing our model to be closer to real world applications.

4.7 Conclusions and Future Work

We have extended an existing mechanisation of x86-assembly created by Jensen et al. to support synchronous interrupts. Jensen's model is expressive enough to reason about mutable code and we stay true to this design philosophy by storing the IDT and all handlers in memory, allowing them to be dynamically updated by the processor. Our extensions to the program logic are also very conservative. By allowing the memory points-to predicate to state that certain memory is unmapped (and not only what it contains), we obtain a logic that is expressive enough to verify programs that use synchronous interrupts. We believe that this is a testament not only to the validity of our design decisions, but also of the quality of the original mechanisation.

4.8 Acknowledgements

We would like to thank Jonas Jensen for fruitful discussions.

Appendix A

Set theory vs Type Theory

This thesis is motivated by the fact the constructive theories are more suitable for working with programming languages and that constructive mathematics are to be preferred.

On the other hand, it is not our intention to make religious claims about what theory is the best. Our opinion is that only practical motivations should lead research in computer science and that philosophical issues should be left out to philosophers.

Therefore, we try to unbiasedly explain the differences between classical and constructive mathematics by giving the reader some crunchy details that can be brod over. Our hope is that an outsider can hone its perception of the problems in formalising programming languages in one theory rather than another.

A.0.1 Set theory

Set theory is a formal system consisting of a set of axioms sitting on a predicate logic. Formally, the grammar of a predicate logic consists of a set of terms inductively defined as

$$t := x \mid f(\vec{t})$$

where x is a set of variables and f is a set of functions symbols with variable arity. Constant symbols are 0-ary function symbols. Formulas are inductively defined as

$$A, B := A \wedge A \mid \forall x.A \mid A \rightarrow B \mid \neg A \mid t = t \mid R(\vec{t})$$

where $\wedge$, $\forall$, $\rightarrow$, $\neg$ are the usual logical operators and where $=$ is an equality on terms and R is a set of relational symbols. The substancial difference about terms and formulas is that the former have values of some kind as domain and codomain, whereas formulas carry the truth of statements such as equality or relational symbols.

ZFC is widely accepted version of set theory and is defined by setting f as the empty set, R by the set of relational symbols containing $\in$ – a binary relation on terms – and by adding a set of axioms ruling the interaction of $\in$ with the other formulas. Before going into some of the most important axioms and attentive reader will immediately notice a small peculiarity arising from the definitions. In fact, not the grammar, nor the rules, prevent the user of the logic to write statements such as $x \in x$. The provability or non-provability of this statement is irrelevant. What is relevant is that it is even possible to write such a "query" into the formal system. We are going back to these problem shortly.

ZFC postulates the existence of a particular kind of object called *set*. Formally, $\exists x.(x = x)$. This is the *Set Existence* axiom and informally says that the universe is not void. Moreover, any set is *extensional* by means of the Extensionality axiom. This means that two sets are equal if they contain the same elements. Formally,

$$\forall x, y, z.(z \in x \leftrightarrow z \in y) \to x = y \tag{A.1}$$

Another central axiom is the Comprehension Scheme Axiom which allows for creation of new sets. If A is a set and P is a predicate logic formula, i.e. *predicate*, on A then the set $\{x : A \mid P(x)\}$ exists and is precisely the set containing all and only those elements of A which satisfy P.

One might try to formalise this axiom as $\exists y.\forall x.x \in y \leftrightarrow \phi$. However, if one chooses ϕ to be $x \notin x$ then the above becomes $\forall x.x \in y \leftrightarrow x \notin x$ for an existing y. Therefore, the statement $y \in y \leftrightarrow y \notin x$ holds leading to the Russel Paradox.

In ZFC, the Comprehension Scheme axiom is formulated by restricting the elements of the new set to a subset of another given set. We defer the reader to the literature [Kun80] for a precise formulation.

Other axioms permit to constructs sets out of other sets, such as the cartesian product, the union and the power set. Another important axiom is the one that postulates the existence of the set of natural numbers $\mathbb{N}$ (Infinity Axiom). The last axiom is the Axiom of Choice which states that whenever w'se have a set of non-empty sets we can pick one element out of any of these sets. The axiom of choice implies $A \vee \neg A$ for all A thus making the logic classical.

In logic, if S is a list of formulas, S is *consistent* iff for no formula ϕ it holds that both ϕ and $\neg\phi$ is derivable from S. If S is inconsistent if for all formulas ϕ, $S \vdash \phi$.

A.0.2 Type theory

Type theory can be read both from a computer science and from a logic point of view via the Curry-Howard correspondence which is the observation that "a proof is a program and the formula it proves is a type for the program". The correspondence has been the starting point of a large spectrum of new research after its discovery, leading in particular to a new class of formal systems designed to act both as a proof system and as a typed functional programming language. Martin-Löf's intuitionistic type theory [ML84; NPS90; NPS00] and Coquand's Calculus of Constructions [CH] underpinned proof-assistants like Agda [Nor07] or Coq [Tea12].

In type theory the basic judgment is formed by a context Γ, a proof t and a formula A. Syntactically, $\Gamma \vdash t : A$ means A has a proof t using assumptions in Γ. In the empty context, $t : A$ is read as "t is an element of A". Moreover, in type theory every object has a type. Thus, a type A has type $\mathcal{U}$, the universe of types. Since being the universe a type it must have a type too, but if we allowed the "universe of all types" $\mathcal{U} : \mathcal{U}$, as in set theory, then also the empty type would be in there leading to the Russel Paradox.

Thus, in type theory, we have a hierarchy of *types*

$$\mathcal{U}_1 : \mathcal{U}_2 : \mathcal{U}_3 \cdots$$

where every universe $\mathcal{U}_i$ is an element of the next universe $\mathcal{U}_{i+1}$. Furthermore, functions are first-class object. The type $A \to B$ represent the function space of all total functions. More generally, for a *family of types* $x : A \vdash B : \mathcal{U}$, we can form the *dependent product* $\Pi(x : A).B$. Similarly, we can form the *dependent sum* or constructive existential $\Sigma(x : A).B$.

Under the Curry-Howard correspondence, the types $A \to B$, $\Pi(x : A).B$ and $\Sigma(x : A).B$ can be equivalently read as formulas. Because of the strength of constructivism curiously enough some type theoretic counterparts of the axioms of set theory can be proven within type theory. For instance the axiom of choice can be proven in its type theoretic reformulation:

$$(\Pi(x : A)\Sigma(y : B)R(x, y)) \to (\Sigma(f : A \to B)\Pi(x : A)R(x, f(x))) \tag{A.2}$$

which means there exists a function that takes a proof of $R(x, y)$ for all x and y and constructs f and the proof that f behaves as the relation R.

A.0.3 Relating type theory and set theory

Type theory differs from set theory in many aspects. While set theory sits on top of first-order logic with the additional notion of sets, type theory has its own formal system where the basic notion is the one of *types*. Formally, whereas the basic judgment in set theory is the one of first-order logic $\vdash A$ in type theory the judgment is a relation $vdasht : A$ between a *term* t and a type *type* A. Moreover, in type theory the type – under the Curry-Howard correspondence – is a formula in intuitionistic logic.

The first consequence of this is a different notion of "membership". The type theoretic counterpart of $\vdash t \in A$ is the judgment $\vdash t : A$. Although, this relation might make sense at first sight, it is not really a good intuition to have in the long run as the judgment in type theory $\vdash t : A$ is *not* a formula. As a consequence, once we stated that t is a proof of A, A cannot be disproven. On the other hand, $t \in A$ is a formula in set theory that can be falsified. In trying to explain set theory in type theory one would have to encode the relational symbol $\in$ more appropriately as a map into the universe of types, so that one can ask what is the proof p such that $\vdash p : t \in A$.

The treatment of equality is also different. In set theory the equality is the least relation that contains all pairs (x, x) for x in some set A, whereas in type type theory the equality is a dependent type: a map from the type $A \times A$ to the universe type $\mathcal{U}$. Thus, to prove the identity type is inhabited for two terms we must show a *proof*. However, sometimes we want to prove two terms are "equal by definition". For instance, $2 + 2$ equals 4 by just unfolding the definition of the function $+$. However, using the former notion of equality this could not be done as we would need a proof. To overcome this problem, in type theory we have also a notion of "judgmental equality". In other words, two terms t and t' of type A can be equal up-to β and η reduction under the judgment $\vdash t \equiv t' : A$.

The treatment of the equality, however, is far from being totally understood. Mathematicians have always deemed isomorphic structures to be equal. On the other hand, only in type theory this identification can be made precise. Homotopy type theory (HoTT) [Uni13] is a type theory where types are spaces (sets with points) and identity proofs are paths between two points in the space. The identification of isomorphic structure is made possible by the so called *univalent axiom* which states that the type of equivalent objects is equivalent to the identity type.

In set theory this axiom is inconsistent. This is easily seen by taking two singleton sets with two different elements in them, namely the set $\{1\}$ and the set $\{2\}$. This two sets are clearly isomorphic and thus by univalence they are equal. However, by Extensionality(A.1) they contain the same elements and thus $1 = 2$.

Bibliography

[Age95] Sten Agerholm. "LCF Examples in HOL". In: *Comput. J.* (1995).

[Ahm04] Amal J. Ahmed. "Semantics of Types for Mutable State". PhD thesis. 2004.

[Ahm06] Amal J. Ahmed. "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". In: *ESOP*. 2006.

[AM13] Robert Atkey and Conor McBride. "Productive Coprogramming with Guarded Recursion". In: *ICFP*. 2013, pp. 197–208.

[App+07] Andrew W. Appel et al. "A very modal model of a modern, major, general type system". In: *POPL*. 2007, pp. 109–122.

[App+14] Andrew W. Appel et al. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[Bar+96] F. Bartels et al. *Formalizing Fixed-Point Theory in PVS*. Tech. rep. Universitat Ulm, 1996.

[BB16] Ales Bizjak and Lars Birkedal. "A taste of Categorical Logic". In: (2016).

[BBM14] Aleš Bizjak, Lars Birkedal, and Marino Miculan. "A Model of Countable Nondeterminism in Guarded Type Theory". In: *RTA-TLCA*. 2014, pp. 108–123.

[Ben+10] Nick Benton et al. "Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages". 2010.

[Ben+11] Jesper Bengtson et al. "Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq". In: *ITP*. 2011.

[BHN14] Nick Benton, Martin Hofmann, and Vivek Nigam. "Abstract Effects and Proof-Relevant Logical Relations". In: *POPL*. 2014.

[Bir+12] L. Birkedal et al. "First steps in synthetic guarded domain theory: step-indexing in the topos of trees". In: *LICS*. 2012.

[Bir+16] L. Birkedal et al. "Guarded Cubical Type Theory". In: *TYPES*. 2016.

[Biz+16] A. Bizjak et al. "Guarded Dependent Type Theory with Coinductive Types". In: *FoSSaCS*. 2016.

[Biz16] Aleš Bizjak. "On semantics and applications of guarded recursion". PhD thesis. 2016.

[BJB12] J. Bengtson, J. B. Jensen, and L. Birkedal. "Charge! – A framework for higher-order separation logic in Coq". In: *ITP*. 2012.

[BKV09] Nick Benton, Andrew Kennedy, and Carsten Varming. "Some Domain Theory and Denotational Semantics in Coq". In: *TPHOLs*. 2009.

[BM13] Lars Birkedal and Rasmus Ejlers Møgelberg. “Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes”. In: *LICS*. 2013, pp. 213–222.

[BM15] Aleš Bizjak and Rasmus Ejlers Møgelberg. “A model of guarded recursion with clock synchronisation”. In: *MFPS*. 2015.

[BP] Gavin M. Bierman and Valeria de Paiva. “On an Intuitionistic Modal Logic”. In: *Studia Logica* ().

[BST09] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. “Realizability Semantics of Parametric Polymorphism, General References, and Recursive Types”. In: *FOSSACS*. 2009.

[BST12] L. Birkedal, K. Støvring, and J. Thamsborg. “A Relational Realizability Model for Higher-Order Stateful ADTs”. In: *Journal of Logic and Algebraic Programming* (2012).

[BTY05] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. “Semantics of Separation-Logic Typing and Higher-Order Frame Rules”. In: *LICS*. 2005.

[Cap05] Venanzio Capretta. “General recursion via coinductive types”. In: *Logical Methods in Computer Science* (2005).

[CH] Thierry Coquand and Gerard Huet. “The Calculus of Constructions”. In: *Inf. Comput.* ().

[Chl11] A. Chlipala. “Mostly-automated verification of low-level programs in computational separation logic”. In: *PLDI*. 2011.

[Clo+15] Ranald Clouston et al. “Programming and Reasoning with Guarded Recursion for Coinductive Types”. In: *FoSSaCS*. 2015.

[COB03] Cristiano Calcagno, Peter W. O’Hearn, and Richard Bornat. “Program logic and equivalence in the presence of garbage collection”. In: *Theor. Comput. Sci.* (2003).

[Coh+15] Cyril Cohen et al. “Cubical type theory: a constructive interpretation of the univalence axiom”. In: (2015).

[Cor13] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer’s Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference*. 2013.

[COY07] C. Calcagno, P. W. O’Hearn, and H. Yang. “Local Action and Abstract Separation Logic”. In: *LICS*. 2007.

[DAB11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. “Logical Step-Indexed Logical Relations”. In: *Logical Methods in Computer Science* (2011).

[Dan12] Nils Anders Danielsson. “Operational semantics using the partiality monad”. In: *ICFP*. 2012, pp. 127–138.

[Esc99] M.H. Escardó. “A metric model of PCF”. Laboratory for Foundations of Computer Science, University of Edinburgh. 1999.

[Fen+08] Xinyu Feng et al. “Certifying low-level programs with hardware interrupts and preemptive threads”. In: *PLDI*. 2008.

[Fio96] Marcelo P. Fiore. “Axiomatic Domain Theory”. 1996.

[FP94] Marcelo P. Fiore and Gordon D. Plotkin. “An Axiomatization of Computationally Adequate Domain Theoretic Models of FPC”. In: *LICS*. 1994.

[Gun92] C. A. Gunter, ed. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.

[HHP93] Robert Harper, Furio Honsell, and Gordon D. Plotkin. "A Framework for Defining Logics". In: *J. ACM* (1993).

[Hyl91] J. Martin E. Hyland. "First steps in synthetic domain theory". In: *Category Theory*. 1991, pp. 131–156.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. "BI as an Assertion Language for Mutable Data Structures". In: *POPL*. 2001.

[JBK13] J. B. Jensen, N. Benton, and A. J. Kennedy. "High-Level Separation Logic for Low-Level Code". In: *POPL*. 2013.

[Jun+15] Ralf Jung et al. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *POPL*. 2015.

[Ken+13] Andrew Kennedy et al. "Coq: the world's best macro assembler?" In: *PPDP*. 2013.

[Kun80] Kenneth Kunen. *Set theory : an introduction to independence proofs*. Studies in logic and the foundations of mathematics. Amsterdam, Lausanne, New York: Elsevier science, 1980. ISBN: 0-444-86839-9. URL: `http://opac.inria.fr/record=b1117475`.

[LSS84] Jacques Loeckx, Kurt Sieber, and Ryan D. Stansifer. *The foundations of program verification*. Wiley-Teubner series in computer science. Stuttgart, Germany: B. G. Teubner Chichester New York, 1984. ISBN: 0-471-90323-X.

[Luo90] Zhaohui Luo. "An Extended Calculus of Constructions". PhD thesis. 1990.

[MG07] Magnus O. Myreen and Michael J. C. Gordon. "Hoare Logic for Realistically Modelled Machine Code". In: *TACAS*. 2007.

[Mil72] Robin Milner. *Logic for Computable Functions: Description of a Machine Implementation.* Tech. rep. Stanford, CA, USA, 1972.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Springer-Verlag, 1971. ISBN: 0387900357.

[ML84] Per Martin-Löf. *An intuitionistic theory of types*. 1984.

[MP08] C. McBride and R. Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (2008).

[Møg06] Rasmus Ejlers Møgelberg. "Interpreting Polymorphic FPC into Domain Theoretic Models of Parametric Polymorphism". In: *ICALP*. 2006.

[Møg09] Rasmus Ejlers Møgelberg. "From parametric polymorphism to models of polymorphic FPC". In: *Mathematical Structures in Computer Science* (2009).

[Møg14] Rasmus Ejlers Møgelberg. "A type theory for productive coprogramming via guarded recursion". In: *CSL-LICS*. 2014.

[Nak00] Hiroshi Nakano. "A modality for recursion". In: *LICS*. 2000, pp. 255–266.

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. 2007.

[NPS00] B. Nordström, K. Petersson, and J. M. Smith. *Martin-Löf's Type Theory*. 2000.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *CSL*. 2001.

[Pho91] Wesley Phoa. "Domain Theory in Realizability Toposes". PhD thesis. 1991.

[Pit96] Andrew M. Pitts. "Relational Properties of Domains". In: *Inf. Comput.* 127.2 (1996), pp. 66–90.

[Plo77] G. Plotkin. "LCF considered as a Programming Language". In: *Theoretical Computer Science* 5.3 (1977), pp. 223–256.

[Plo93] Gordon D. Plotkin. "Type Theory and Recursion (Extended Abstract)". In: *LICS*. 1993.

[PMB15] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. "A Model of PCF in Guarded Type Theory". In: *Electr. Notes Theor. Comput. Sci.* (2015).

[Pol94] Robert Pollack. "The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions". PhD thesis. Univ. of Edinburgh, 1994.

[PP02] Gordon D. Plotkin and John Power. "Notions of Computation Determine Monads". In: *FOSSACS*. 2002.

[Reg95] Franz Regensburger. "HOLCF: Higher Order Logic of Computable Functions". In: *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*. 1995.

[Reu95] Bernhard Reus. "Program verification in synthetic domain theory". PhD thesis. 1995.

[Reu96] Bernhard Reus. "Synthetic Domain Theory in Type Theory: Another Logic of Computable Functions". In: *TPHOLs*. 1996.

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. 2002.

[Ros86] G. Rosolini. "Continuity and effectiveness in topoi". PhD thesis. University of Oxford, 1986.

[RS04] G. Rosolini and A. Simpson. "Using Synthetic Domain Theory to Prove Operational Properties of a Polymorphic Programming Language Based on Strictness". 2004.

[RS99] Bernhard Reus and Thomas Streicher. "General synthetic domain theory - a logical approach". In: *Mathematical Structures in Computer Science* (1999).

[RY04] Uday S. Reddy and Hongseok Yang. "Correctness of data representations involving heap data structures". In: *Sci. Comput. Program.* (2004).

[SB14] Kasper Svendsen and Lars Birkedal. "Impredicative Concurrent Abstract Predicates". In: *ESOP*. 2014.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Dubuque, IA, USA: William C. Brown Publishers, 1986. ISBN: 0-697-06849-2.

[Sco69] Dana S. Scott. "A Type-theoretical Alternative to ISWIM, CUCH, OWHY". 1969.

[Sco80] D.S. Scott. "Relating Theories of the λ-calculus". In: *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalisms*. Ed. by R. Hindley and J. Seldin. Academic Press, 1980, pp. 403–450.

[Sim02] Alex K. Simpson. "Computational Adequacy for Recursive Types in Models of Intuitionistic Set Theory". In: *LICS*. 2002, pp. 287–298.

[SP77] Michael B. Smyth and Gordon D. Plotkin. "The Category-Theoretic Solution of Recursive Domain Equations (Extended Abstract)". In: *FOCS*. 1977.

[Str06] Thomas Streicher. *Domain-theoretic foundations of functional programming.* World Scientific, 2006.

[Tay91] Paul Taylor. "The Fixed Point Property in Synthetic Domain Theory". In: *LICS*. 1991.

[Tea12] The Coq Development Team. *The Coq Reference Manual, version 8.4*. 2012. URL: `http://coq.inria.fr`.

[tea88] The HOL theorem prover team. *HOL Interactive theorem prover*. 1988. URL: `https://hol-theorem-prover.org`.

[VB08] Carsten Varming and Lars Birkedal. "Higher-Order Separation Logic in Isabelle/HOLCF". In: *Electr. Notes Theor. Comput. Sci.* (2008).

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `http://homotopytypetheory.org/book`, 2013.