Formal Reasoning for Modern Programming Languages

Sergei Stepanenko

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Formal Reasoning for Modern Programming Languages

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Sergei Stepanenko
August 31, 2025

Abstract

Computer programs have been ubiquitous for several decades. So are bugs and errors in computer programs and systems. As the use of programming in critical areas is universal, the consequences of these bugs and errors can be severe, so we want to develop methods to avoid them. This is one of the pragmatic reasons to study the semantics of programming languages; we want to be able to establish properties that we expect from computer programs and prove that the properties are satisfied. To achieve that, we focus on two properties of programming languages semantics. First, they need to be rigorous; that is, we want to study semantics from the mathematical perspective, so that demonstration of properties of computer programs are not just verbal claims but mathematical proofs. Second, semantics of programming languages should be practically usable — it should be feasible to verify interesting properties, and the proofs of those properties should not be burdensome or practically unachievable.

In this thesis, we address the rigorous and practical aspects of programming languages semantics. In particular, we study the semantics of generalized algebraic datatypes, a feature that was adopted by Haskell and OCaml programming languages, and construct a model of the example language that employs this feature. We also investigate the semantics of languages with context-dependent effects, which are also present in Haskell and OCaml to some extent. To achieve that, we extend the existing theory of Guarded Interaction Trees to support context-dependent effects, and demonstrate its usage when combined with two popular flavors of context-dependent effects: call with current continuation (colloquially call/cc) and delimited continuations. Lastly, we simplify the construction of solutions to a class of recursive domain equations and use the simplification to mechanize the solver in the Rocq Prover.

Resumé

Computerprogrammer har været allestedsnærværende i flere årtier. Det samme gælder fejl i computerprogrammer og systemer. Da programmering er universelt anvendt i kritiske områder, kan konsekvenserne af disse fejl være alvorlige, og vi ønsker derfor at udvikle metoder til at undgå dem. Dette er en af de pragmatiske grunde til at studere programmeringssprogssemantik; vi vil kunne fastlægge de egenskaber vi forventer af computerprogrammer, og bevise at egenskaberne er opfyldt. For at opnå dette fokuserer vi på to egenskaber for programmeringssprogssemantik. For det første skal de være stringente; det vil sige, at vi vil studere semantik fra et matematisk perspektiv, så demonstrationen af computerprogrammers egenskaber ikke kun er verbale påstande, men matematiske beviser. For det andet skal programmeringssprogssemantik være praktisk anvendeligt – det skal være muligt at verificere interessante egenskaber, og beviserne for disse egenskaber må ikke være besværlige eller praktisk umulige at opnå.

I denne afhandling behandler vi de stringente og praktiske aspekter af programmeringssprogsemantik. I særdeleshed studerer vi semantikken for generaliserede algebraiske datatyper, en feature anvendt af Haskell og OCaml, og konstruerer en model af et eksempel på et sprog, der anvender denne feature. Vi undersøger også semantikken i sprog med kontekstafhængige effekter, som også findes i Haskell og OCaml i et vist omfang. For at opnå dette udvider vi den eksisterende teori for Guarded Interaction Trees til at understøtte kontekstafhængige effekter og demonstrerer dens anvendelse, når den kombineres med to populære varianter af kontekstafhængige effekter: call with current continuation (også kendt som call/cc) og delimited continuations. Endelig forsimpler vi konstruktionen af løsninger til en klasse af rekursive domæneligninger og bruger forsimplingen til at mekanisere solveren i Rocq.

Acknowledgments

This Ph.D. was a great experience for me and I would like to express my gratitude to the people responsible for that.

First and foremost, I would like to thank my supervisor, Lars Birkedal. Lars was always supportive and encouraging, and he patiently tolerated my shenanigans throughout these years. I am truly indebted to him for his exceptional guidance. His many insightful suggestions have pointed me in new interesting directions during these years, and his guidance has shaped not only this thesis but also my growth as a person.

Although Amin Timany was not my official supervisor, he was an extraordinary mentor. He was always available to answer questions, offer advice, and provide thoughtful feedback. His office door was always open, and his expertise and comments on a wide range of topics have been tremendously helpful.

I also would like to thank my bachelor's supervisor, Vitaly Bragilevsky. Vitaly introduced me to the world of theoretical computer studies and encouraged me to pursue a Ph.D. in the semantics of programming languages.

My gratitude extends to all the members of the LogSem group. I am also thankful to my friends and colleagues at Aarhus University, who made my time there not only productive but also truly enjoyable. In particular, I would like to mention Jonas, Alejandro, Zongyuan, Jean, Simon and Simon, Daniel, June, Armaël, Aïna, Egor, Philipp and Philipp, Maxime, Irfansha and Arnaud.

I am very grateful to my coauthors — Filip, Jean, Jon, Zongyuan, Aslan, Emma, Dan, Amin and Lars. I learned so much from our collaborations and look forward to continuing to learn in future projects.

During the last year, I spent two months at the University of Groningen, hosted by Dan Frumin in the Fundamental Computing group. Dan and the other members of the group made this visit a wonderful experience. These two months were really good, and I started liking working with math that is not necessarily formalized in proof assistants.

I would like to thank my entire family and especially my mother, my grandmother, my aunt, uncle, and cousin for their support and care before and during my Ph.D. Special thanks go to Giovanna, who made the last two years of my Ph.D. truly amazing.

Finally, I would also like to thank my friends, both those from before my time in Aarhus, who managed to keep our friendships alive despite the distance, and those I met in Aarhus: Misha, the three Ivans, Alexander, Denis, Kay, Misha, Erik, Oliver, Dennis, Marta, Isabell, Anders, Katrine, Line, Charlotte, Emīls, and many more.

Sergei Stepanenko,
Aarhus, August 31, 2025.

Contents

Part I

Overview

CHAPTER 1

Introduction

This thesis is a compilation of works that I coauthored during my Ph.D. All of the topics are connected by one common cornerstone: modeling of features encountered in modern programming languages. In the following, we provide an overview of the content.

1.1 Outline of the Dissertation

In this section, we provide an outline of the different parts of this thesis.

This thesis is divided into two parts, with Part I providing an overview of the included publications. Part I consists of one chapter. This chapter provides the background for this dissertation. After the outline, in Section 1.2, we provide a general context for the work presented in this thesis. Next, Section 1.3 contains a cursory introduction to the semantic studies of programming languages. This section targets a broad audience and is supposed to provide a bird's eye view on semantic methods. In particular, we motivate studying programming languages from the semantics perspective, and demonstrate it on a small example language. Next, in Section 1.4 we provide a more domain-specific introduction to the topics studied in the thesis. In particular, this section targets readers working with the semantics of programming languages but not necessarily familiar with the features we study in this work. In particular, we provide a separate and more targeted overview of the three topics studied in Part II. In Sections 1.4.1 to 1.4.3 we describe the problems studied in the corresponding chapters of Part II, discuss the state of the art, the research questions studied, and the results obtained. In Section 1.5 we briefly discuss potential directions for future work. The chapter concludes with Section 1.6, where we include a list of publications used in this thesis and give a statement of personal contributions.

Part II is based on three published papers to which I contributed during my Ph.D. Part II contains three chapters, one per paper.

In Chapter 2, we provide reasoning principles for a feature commonly used in modern functional programming languages, *Generalized Algebraic Datatypes* (GADTs), which are used to provide stronger guarantees about programs that can be

encoded using a type system and type-checked by compilers. We show a language that allows one to encode GADTs and provide models for this language that allow one to demonstrate properties of programs utilizing GADTs. In doing so, we also construct a novel model that mixes both syntactical and semantic components. For an exposition and historical overview, refer to Section 1.4.1.

In Chapter 3 we extend the *Guarded Interaction Trees* (GITrees) framework with context-dependent effects, such as `call/cc` and `shift/reset`. To demonstrate the extension, we provide models of two languages: one with `call/cc` and one with `shift/reset`, and show that the models are sound and adequate with respect to the intended operational semantics. Our extension is backward compatible with previous work, which allows one to reason about programming languages that contain both context-dependent effects and more widespread effects, such as higher-order state or I/O. For an exposition and historical overview, refer to Section 1.4.2.

In Chapter 4 we consider a simplification of constructing solutions to *Recursive Domain Equations* (RDEs). In particular, we simplify the existing approach of solving RDEs in the category of sheaves over ordinals to the category of presheaves over ordinals. Presheaves are more amenable to mechanization than sheaves, which we demonstrate by mechanizing the solver in the Rocq Prover[1]. For an exposition, refer to Section 1.4.3.

1.2 Context of this Work

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> Edsger W. Dijkstra

In this section, we provide a broad overview on a few sub-areas of semantics of programming languages and the need for semantic models of programming languages.

Challenges of verifying software. Programming languages have continuously evolved over many decades. From simple languages used just to describe numerical operations, we went to programming languages that utilize higher-order programs, provide intricate control flow operations, and use sophisticated type systems to statically rule out entire classes of bugs. The programs using these features are often intricate, and while they allow us to express a lot of functionality in just a few lines of code, it is often nontrivial to use them. Moreover, new features keep getting introduced and, with them, the expressivity of programming languages improves on par with the complexity of programs written using these features. Thus, it becomes increasingly important to have a proper understanding of the new language features to be able to reason about the correctness of the programs that use them.

[1] Previously known as the Coq Proof Assistant.

Verifying correctness becomes even more challenging when the new features are not simply incremental but have nontrivial interactions with existing constructions, which is often the case. Some examples include features studied in this thesis. For example, both Haskell and OCaml support *generalized algebraic datatypes*. While allowing to impose additional restrictions on elements of certain datatypes, thus ruling out undesired behaviors, this feature is mostly studied from the implementation viewpoint, with the majority of works focusing on type-checking algorithms, while reasoning about programs using this feature was largely not studied. Another example comes from advanced control flow manipulations. OCaml 5.0 introduced effect handlers, which allow users to define new effects that can inspect the rest of the program. In particular, effect handlers allow us to implement powerful features, such as cooperative concurrency or backtracking, using existing language constructions, instead of providing them as language primitives. While greatly increasing the expressive power of the language, it also means that reasoning about the code with effects becomes complicated. The Glasgow Haskell Compiler community, also interested in effect systems, introduced *delimited continuations* into the core language of GHC, starting from version 9.6.1. The goal of this addition is to allow users to implement nontrivial effects on top of the existing language without modifying the Glasgow Haskell Compiler itself or writing plugins for it.

Verifying that programs behave as expected is surprisingly difficult. There are multiple approaches to this problem. One of them is to test programs on some test cases and validate that the programs behave as expected. This is often a straightforward path, and testing programs on different inputs is widely used in industry. For example, take the following program in C, which computes the greatest common divisor of two numbers.

```
int gcd(int a, int b) {
  int i = a;
  int j = b;
  int t;
  while (j != 0) {
    t = j;
    j = i % j;
    i = t;
  }
  return i;
}
```

We can test it with different integers, and check that it returns the expected results. This would provide some guarantees, but the guarantees would only provide assurance for the covered cases and not guarantee the absence of bugs. Moreover, testing programs is often not practically feasible. For example, we can consider the following example in OCaml:

```
type 'a t = 'a list =
| nil
| cons of 'a * 'a list

let rec map (f: 'a -> 'b) (l: 'a list): 'b list =
match l with
| nil = []
| (cons x xs) = cons (f x) (map f xs)
```

It is an example of a higher-order program, which, given a list, returns a new list, computed from the initial one with a user-supplied function.

Even if we fix `'a` and `'b` to 32-bit signed integers, the amount of potential tests for this program grows significantly, as one needs to test it with different functions on integers. It becomes even more challenging if we consider not just pure numerical functions but functions that can invoke effects. For example, we can call `map` with a function that raises an exception — `map (fun _ -> failwith "Oh, no!") (cons 1 nil)`. Or, alternatively, we can allocate a reference and have the supplied function access the content of the reference when called. If we drop the assumption that `'a` and `'b` are types of 32-bit signed integers, we have even more cases to cover, as now `'a` and `'b` can be any types, including ones defined by users. This illustrates the practical challenges of verifying programs utilizing complex behavior.

The semantic perspective on correctness. Another technique is to formally verify that programs or selected features behave as expected. While often requiring more effort, this approach allows one to provide much stronger guarantees and is especially appealing to critical software. In this work, we study programming languages from the *semantic* perspective. By semantic perspective, we mean that we describe the intended meaning of different programming languages' constructions mathematically and reason about the assigned meaning; *i.e.* we provide a model for a programming language. In the following paragraphs, we provide an exposition to semantic reasoning about programming languages by showing two different semantics of a minimal programming language and discussing them.

An alternative view on semantics. In this work, we study semantics from the verification perspective. That is, we use the constructed models to reason about the correctness of written programs. However, it is worth noting that there is another perspective — extending programming languages. By studying models of programming languages, we can derive new gadgets that can be safely added to programming languages. In particular, PCF, the programming language for computable functions [98], which greatly influenced the subsequent design and development of the ML family of languages, takes its origin from semantics of the untyped lambda calculus. The study of PCF, and one of its semantic models, the logic for computable functions (LCF), gave rise to various extensions of the language. We do not use semantics of

identifiers	Ident	::=	$x, y, z, \ldots$
expressions	Expr $\ni e$	::=	$n \mid e_1 + e_2 \mid e_1 - e_2 \mid x$
statements	Stm $\ni s$	::=	$x := e \mid s_1 ; s_2 \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{skip}$
programs	Prog $\ni p$	::=	$\cdot \mid s ; p$

Figure 1.1: Syntax of IMP.

programming languages from this perspective in this thesis, but it is important to highlight that there are other motivations to study semantic models.

Mechanized semantics. Another important aspect of the work in this thesis is that we focus on *mechanized* semantics. Models of programming languages are often intricate to define, as we will briefly show, and have a lot of easy-to-miss details. To address that, semantic studies are formalized in *proof assistants*. A proof assistant is a program that allows one to formalize and reason about mathematics while having a machine to verify the correctness of proofs. To increase reliability, all of the works in this thesis are mechanized in the Rocq Prover. In particular, providing mechanized semantics for programming languages is one of the main motivations of Chapter 3 and Chapter 4.

1.3 Background

By studying languages from the semantic perspective, we formally show that software has certain behaviors. For the running example, let us consider a simple language that illustrates the advantages and challenges of reasoning about the semantics of programming languages.

Our first example is a simple language called IMP [100] shown in Figure 1.1. It is an imperative language that can be viewed as a simplified version of Pascal or ALGOL 60.

To reduce ambiguity, we use blue for various language operators and keywords and black for the usual mathematical operations. This language employs some simple operations on integers, allowing one to specify integer literals, perform addition and subtraction. In addition, the language has control operations that allow one to perform branching and iteration. To simplify the language, we use integers instead of boolean. The $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ statement executes the second branch if e computes to a negative integer or zero, and the first branch otherwise. The $\mathsf{while}\ e\ \mathsf{do}\ s$ statement checks if e reduces to a negative integer or zero, and, if so, skips the body of the loop and continues execution. If e computes to a positive integer, $\mathsf{while}\ e\ \mathsf{do}\ s$ executes s, and loops. On top of that, the language has a state given by a mapping from identifiers to integers. Users can allocate or modify the state with assignment statements $x := e$, and retrieve the content of existing mappings by referring to their identifiers in expressions. Operations on integers can use values stored in memory

cells by using the identifiers corresponding to them. The last statement, skip, does nothing and the machine simply proceeds with the rest of the program. A complete program in IMP is a list of statements.

Operational semantics. The first style of semantics we consider is *operational* semantics, shown in Figure 1.2. Operational semantics has a long history, dating back to the earliest studies of programming languages[2]. We can consider some machine that executes this program and describe how this machine is going to execute programs written in this language, operation by operation, ignoring the implementation of such a machine.

In this particular example, we consider a machine that maintains a mapping from identifiers to integers, where ε denotes the starting empty state and $\sigma[i \mapsto n]$ denotes a state of the machine that was extended with a new mapping that assigns n to the identifier i. For the full machine configuration, we consider a pair of the program we intend to execute and the current state.

state	$\text{State} \ni \sigma$	$::= \ \varepsilon \mid \sigma[i \mapsto n]$
configuration	$\text{Conf} \ni c$	$::= \ (p, \sigma)$

Every execution step of this machine is going to modify the state based on the current statement of the program. We can describe this behavior by defining two relations. The step relation, $\rightarrow \subseteq \text{Conf} \times \text{Conf}$, specifies the behavior of the complete machine, *i.e.* $(p_1, \sigma_1) \rightarrow (p_2, \sigma_2)$ means that the machine makes a step from a state σ_1 and a program p_1 to some new state σ_2 and some new program p_2. The evaluation relation, $\downarrow \subseteq (\text{Expr} \times \text{State}) \times \mathbb{Z}$, is a relation that specifies how expressions are evaluated into concrete integers. The rules are presented in a standard format for operational semantics, where propositions above the horizontal line are a condition for the relation below the line to hold. For example, consider the following clause:

$$\frac{(e, \sigma) \downarrow n \qquad n \leq 0}{(\textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2 \,;\, p, \sigma) \rightarrow (s_2 \,;\, p, \sigma)}$$

The clause describes the machine's behavior when executing the if e then s_1 else s_2 statement, when the condition given by e does not hold. That is, if e evaluates to a non-positive integer, the machine starts to execute the second branch.

In addition, we abbreviate the reflexive-transitive closure of $\rightarrow$ as $\rightarrow^*$. We consider a program to be terminated if we have exhausted the list of statements, *i.e.* evaluated to $(\cdot, \sigma)$.

Essentially, we specified the behavior of a partial evaluator for programs written in IMP. The partiality comes from the fact that IMP has dynamic scope. That is, nothing prevents us from writing a program that tries to access some identifier that has not been assigned with an expression. When it happens, we say that the operational semantics gets stuck. Next, we proceed with a discussion on some properties of programs implemented in IMP.

[2] Plotkin [99] provided a historical origin of operational semantics.

$$\frac{}{(n,\sigma)\downarrow n} \qquad \frac{\sigma[i\mapsto n]}{(i,\sigma)\downarrow n} \qquad \frac{(j,\sigma)\downarrow n \qquad i\neq j}{(j,\sigma[i\mapsto m])\downarrow n} \qquad \frac{(e_1,\sigma)\downarrow n_1 \qquad (e_2,\sigma)\downarrow n_2}{(e_1+e_2,\sigma)\downarrow(n_1+n_2)}$$

$$\frac{(e_1,\sigma)\downarrow n_1 \qquad (e_2,\sigma)\downarrow n_2}{(e_1-e_2,\sigma)\downarrow(n_1-n_2)}$$

$$\frac{(e,\sigma)\downarrow n \qquad n>0}{(\textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2\,;p,\sigma)\to(s_1\,;p,\sigma)} \qquad \frac{(e,\sigma)\downarrow n \qquad n\leq 0}{(\textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2\,;p,\sigma)\to(s_2\,;p,\sigma)}$$

$$\frac{(e,\sigma)\downarrow n}{(i:=e\,;p,\sigma)\to(p,\sigma[i\mapsto n])} \qquad \frac{(e,\sigma)\downarrow n \qquad n>0}{(\textsf{while } e \textsf{ do } s\,;p,\sigma)\to(s\,;\textsf{while } e \textsf{ do } s\,;p,\sigma)}$$

$$\frac{(e,\sigma)\downarrow n \qquad n\leq 0}{(\textsf{while } e \textsf{ do } s\,;p,\sigma)\to(p,\sigma)} \qquad \frac{}{(\textsf{skip}\,;p,\sigma)\to(p,\sigma)}$$

Figure 1.2: Operational semantics of IMP.

```
mult(e1, e2, r) ≜
  r := 0;
  t := e2;
  if t then
    while t do
      r := r + e1;
      t := t − 1;
  else
    while 0 − t do
      r := r − e1;
      t := t + 1;
```

Figure 1.3: Example program in IMP.

Formal reasoning about the correctness of programs. The operational semantics presented in Figure 1.2 provides the mathematical framework needed to formally state and prove properties of programs written in IMP. In particular, we are interested in *correctness*: given a program specification that describes the expected outcome of execution, we can formally verify that the program satisfies this specification. As an illustration, consider the program in Figure 1.3.

Since IMP does not support function definitions, we treat $\mathsf{mult}(e_1,e_2,r)$ as syntac-

tic sugar for a sequence of instructions. The intended meaning of this program is to compute the product $e_1 * e_2$ and store the result in the state, accessible by the identifier r.

Under the operational semantics, verifying this program amounts to establishing the following lemma:

Lemma 1.3.1.

$$(e_1, \sigma) \downarrow n \wedge (e_2, \sigma) \downarrow m \implies (\mathsf{mult}(e_1, e_2, r)\,;p, \sigma) \to^* (p, \sigma[r \mapsto n * m, t \mapsto 0])$$

This specification comes with certain assumptions. First, e_1 and e_2 are arbitrary expressions, but we assume that both evaluate to concrete integers n and m in the initial state σ. Only under this assumption does it make sense to state that the result of the multiplication is $n * m$.

We now verify that mult satisfies this specification.

Proof. We prove the lemma by constructing an explicit execution sequence. Two cases arise depending on the value of m: either $m > 0$ or $m \leq 0$. Since the argument is almost analogous, we show only the second case $m \leq 0$.

The first few steps execute two assignments followed by the conditional branching, yielding:

$$(\mathsf{mult}(e_1, e_2, r)\,;p, \sigma) \to^* \left(\left[\begin{array}{l} \mathsf{while}\ 0 - t\ \mathsf{do} \\ \quad r := r - e_1\,; \\ \quad t := t + 1\,; \end{array} \right] ;p, \sigma[r \mapsto 0, t \mapsto m] \right)$$

It remains to show that the loop computes the expected result. For this, we prove a stronger statement: the loop evaluates correctly for any initial accumulator value a. Formally, for arbitrary a:

$$\left(\left[\begin{array}{l} \mathsf{while}\ 0 - t\ \mathsf{do} \\ \quad r := r - e_1\,; \\ \quad t := t + 1\,; \end{array} \right] ;p, \sigma[r \mapsto a, t \mapsto m] \right) \to^* (p, \sigma[r \mapsto a + m * n, t \mapsto 0])$$

We prove this by induction on $|m|$.

Base case. If $|m| = 0$, then $m = 0$ and the loop body is never executed. The result is immediate.

Inductive step. Assume the claim holds for $|m| = m' + 1$ for some $m' \geq 0$, *i.e.* we assume that the inductive hypothesis (IH) holds for any a.

$$\left(\left[\begin{array}{l} \mathsf{while}\ 0 - t\ \mathsf{do} \\ \quad r := r - e_1\,; \\ \quad t := t + 1\,; \end{array} \right] ;p, \sigma[r \mapsto a, t \mapsto -m'] \right) \to^* (p, \sigma[r \mapsto a - m' * n, t \mapsto 0]) \qquad \text{(IH)}$$

We fix some a', and proceed with the inductive case. For $|m| = m' + 1$, the loop condition $0 - t$ evaluates to $m' + 1 > 0$, so one iteration of the loop is executed: r decreases by n and t increases by 1. It remains to show that the rest of the loop behaves as expected:

$$\left(\left[\begin{array}{l} \mathsf{while}\ 0 - t\ \mathsf{do} \\ \quad r := r - e_1\,; \\ \quad t := t + 1\,; \end{array} \right] ;p, \sigma[r \mapsto a' - n, t \mapsto -m'] \right) \to^* (p, \sigma[r \mapsto a' - (m' + 1) * n, t \mapsto 0])$$

This follows immediately from the induction hypothesis (IH), instantiated with $a = a' - n$. □

In summary, we defined a mathematical model that faithfully captures the behavior of the programming language. Within this model, verification of programs such as mult is no longer a verbal claim but a rigorous mathematical proof. If the model is adequate and the proof is sound, we gain stronger confidence in the correctness of the software. For example, the proof of correctness for mult shows that, under the stated assumptions, it *always* produces the expected result according to the operational semantics.

Although simple, this example illustrates two key aspects of semantic reasoning about programs. First, semantics makes it possible to establish strong, general guarantees: our proof is not limited to the program mult in isolation, but shows that mult behaves correctly when composed with any remaining program p. Second, even in this small case, the proof involves many details that are easy to overlook. This highlights the value of mechanized proofs, which help to ensure reliability in the study of programming language semantics.

Relational reasoning. While the correctness example shows how operational semantics supports reasoning about programs, correctness is not the only property of interest. A central theme of this thesis is *relational* reasoning, where we compare two programs to establish that one can safely replace the other. For instance, suppose that we introduce an alternative implementation of mult, mult', which may be more efficient but less straightforward to implement. To justify using mult' instead of mult, we must prove a *refinement*: *i.e.*, that mult and mult' behave identically in every context, so that no surrounding program can distinguish them.

This raises the question of what observations are available when running a program in IMP. Because the language has dynamic scoping, executions may get stuck if a variable is accessed, while it is undefined. Moreover, programs may diverge because of the presence of loops. Finally, we can observe whether a program terminates, and, if so, inspect the resulting state by running further programs that query variables. Thus, any difference in how two programs compute internally is irrelevant as long as they either both diverge or both terminate with the same resulting state. Consequently, refinement reduces to comparing programs only with respect to their termination behavior and final states, since no other observations are visible to distinguish them.

More formally, observations can be characterized using program contexts with a single hole, into which the programs under comparison are inserted. Consider two lists of statements C_1 and C_2 of IMP. Together, they represent a context for a program p between them. By changing C_1, we can modify the state in which p starts its execution. Analogously, by picking different lists for C_2, we can test the content of identifiers to determine the state after p.

Returning to the original problem, given programs mult and mult', we can test if the combined program $C_1;\mathsf{mult}'(e_1,e_2,r);C_2$ terminates if $C_1;\mathsf{mult}(e_1,e_2,r);C_2$ terminates. Less formally, by considering all surrounding contexts, we enforce that mult' requires at most the same information as mult, and provides at least the same information as mult. This extensional notion, first presented in James Hiram Morris

[63], is commonly called *contextual refinement*, and its symmetrical closure is considered to be one of the standard notions of program equivalence. In Chapter 2, we employ it to study a language that supports a certain feature in its type system.

Other flavors of semantic models. The operational semantics introduced earlier is only one possible way to model the language. Its main appeal lies in its simplicity: it can be defined and reasoned about within a relatively modest mathematical framework. At the same time, operational semantics becomes cumbersome for larger programs, since it requires spelling out numerous administrative execution steps and maintaining detailed state information. And while it can be understood as an idealized machine executing programs, it remains conceptually close to an actual evaluator, rather than offering a more abstract view of program meaning. That is the reason why, instead of using the operational semantics directly for showing program properties, it is beneficial to either use logical systems on top of the operational semantics, or use a different semantic model.

Denotational semantics. An alternative to the operational semantics discussed previously is *denotational* semantics. Rather than describing how an idealized machine executes programs in IMP, denotational semantics interprets the syntactic components of IMP as mathematical objects. Formally, it is given by a mathematical universe of denotations (the *domain* of the interpretation) $\mathbb{D}$ together with a family of interpretation functions for the syntactic categories of IMP: *e.g.*, $[\![-]\!]_{\mathrm{Expr}} : \mathrm{Expr} \to \mathbb{D}$, $[\![-]\!]_{\mathrm{Stm}} : \mathrm{Stm} \to \mathbb{D}$, and $[\![-]\!] : \mathrm{Prog} \to \mathbb{D}$. In contrast to operational semantics, denotational semantics is *compositional*: the meaning of a program is determined solely by the meanings of its subconstructs. For example, if a program uses mult, we can replace $[\![\mathsf{mult}(e_1, e_2, r)]\!]$ with any denotationally equal program without changing the overall meaning. Another key feature is that $\mathbb{D}$ can be chosen independently of the particular language. This allows a single domain to serve as the semantic universe for multiple languages, enabling reasoning about their interactions within a common framework.

We do not provide an example of denotational semantics for IMP, and instead refer the interested readers to more complete and comprehensive materials. Denotational semantics was introduced by Scott and Strachey as a mathematically rigorous alternative to operational semantics of programming languages [109]. A canonical reference for the underlying theory is the book by Abramsky and Jung [1], which develops domain theory as the basis for denotational models.

Domain choice. In the previous paragraph, we have left the interpretation domain $\mathbb{D}$ unspecified. In practice, the choice of domain is determined by the constructs of the language under study and the reasoning principles one wants to support. For example, in Chapter 3 we fix a particular domain in order to study a family of languages with related constructs. Later, in Chapter 4, we mechanize a more general framework that enables the definition of domains for feature-rich languages, including those with higher-order functions.

Typed semantics. A well-known technique for increasing software reliability is the use of typed languages. The language we have discussed so far, IMP, is untyped: every program built from its grammar is considered a valid program. Modern programming languages, however, typically include type systems that restrict the class of valid programs, ruling out certain kinds of errors before execution. For example, in Chapter 2 and Chapter 3 we define and study typed languages, each equipped with a type system tailored to the language features under consideration. Introducing a type system for IMP would not provide meaningful insights into these features, hence we leave it untyped. However, it is important to emphasize that the untyped semantics of IMP is only one approach. Type systems can be understood as a way of enforcing semantic properties statically, and thus it is often beneficial to make types explicit in the semantics itself. In such *typed semantics*, each type denotes a set of *semantically valid programs*, allowing reasoning to proceed in a type-directed manner. We adopt this approach for works presented in Chapter 2 and Chapter 3, where we show that well-typed programs also have specific semantic properties.

Relating semantic models. Operational semantics is typically employed to specify the basic behavior of a language and to formulate correctness or relational properties. However, as noted before, for more sophisticated reasoning, it is often useful to work with alternative semantic models that provide stronger principles and abstractions. For example, in Chapter 2 and Chapter 3 we begin with operational specifications of the studied languages, and then develop more expressive models that enable more powerful reasoning, relating their properties back to the operational semantics.

1.4 Contributions

In this section we provide a more focused introduction, intended for readers familiar with the semantics of programming languages or related areas, though not necessarily with the specific topics of this thesis. Here we outline the context and contributions of the dissertation in greater detail, with pointers to the corresponding chapters. Specifically, Section 1.4.1 introduces the material developed in Chapter 2, Section 1.4.2 corresponds to Chapter 3, and Section 1.4.3 previews the results of Chapter 4. In Section 1.5 we briefly discuss potential directions for future work, and then finalize this chapter with a list of publications used in this thesis and give a statement of personal contributions in Section 1.6.

1.4.1 Generalized Algebraic Datatypes

In this section we provide background and an overview of generalized algebraic datatypes (GADTs), a generalization of algebraic datatypes (ADTs). Throughout this section, as well as in Chapter 2, we use a Haskell-like syntax for examples.

Algebraic datatypes are defined by a collection of constructors, each describing a way to build an element of the datatype. Constructors take a number of parameters

(possibly including the type being defined, as in recursive types) and return an element of the datatype. For example, the type of lists with elements of type `a`, shown in Figure 1.4, is given by two constructors: `Nil` and `Cons`. Both return values have type `List a`, with `Cons` additionally taking an argument of type `List a`. Ultimately, the type parameter `a` remains fixed across constructor arguments and result types.

By relaxing this restriction on constructor arguments, we obtain *nested types*, which allow type parameters to vary in the arguments. For example, the type of perfect trees in Figure 1.5 includes a constructor `PBranch` whose argument has type `PTree (a, a)`. This differs from lists, where `Cons` preserves the parameter `a`; here, the type parameter is transformed into a product type. As a result, each application of `PLeaf` doubles the payload, forcing the structure to contain 2^n elements at depth n and ensuring that the constructed tree is perfect.

GADTs extend this idea further: in addition to shaping the argument types, a GADT constructor can *specialize the result type parameters*. This makes it possible to encode richer invariants directly in the type system. For example, perfect trees can be defined more succinctly using GADTs, as in Figure 1.6, where natural numbers at the type level track the height of the tree. Here, the result type records height information: empty trees are assigned the type parameter `Z` (zero), while internal nodes require two subtrees of the same height n and produce a tree of height $n+1$, represented by `S n` (successor). Note that it is not important what types are used for the encoding of natural numbers (or if these types are inhabited) as they are used purely for their shape.

Beyond type invariants, GADTs can also enforce type safety for operations. For example, Figure 1.7 defines an expression datatype indexed by the type of value represented. Constructors refine the result type: for instance, `ExprBool` yields `Expr Bool`. As a result, an evaluator `eval :: Expr a -> a` can safely return a value of the correct type in each branch. In the `ExprApp` case, the types guarantee that the left subexpression is a function and the right subexpression is a suitable argument, allowing direct evaluation without runtime checks. In contrast, with a traditional ADT all constructors would yield a uniform `Expr`, making it impossible to encode at the type level that `ExprApp` expects a function on the left. One would have to rely on runtime checks or unsafe casts, losing type guarantees.

The introduction of GADTs into programming languages is motivated by improved type safety: they allow for more invariants[3] to be expressed directly in the type system, ruling out ill-formed programs at compile time. This expressiveness makes GADTs particularly valuable for implementing type-safe interpreters, compilers, and data structures with strong correctness guarantees, eliminating the need for ad-hoc runtime checks.

[3] For example, "an element of `Expr (Bool -> Bool)` always evaluates to a function from booleans to booleans."

```
data List a where
  Nil :: List a
  Cons :: a -> List a -> List a
```

Figure 1.4: Algebraic datatype of lists.

```
data PTree a where
  PLeaf :: a -> PTree a
  PBranch :: PTree (a, a) -> PTree a
```

Figure 1.5: Nested datatype of perfect trees.

```
data Z
data S x
data PTree' n a where
  PLeaf' :: a -> PTree' Z a
  PBranch' :: PTree' n a -> PTree' n a -> PTree' (S n) a
```

Figure 1.6: Generalized algebraic datatype of perfect trees.

```
data Expr a where
  ExprBool :: Bool -> Expr Bool
  ExprLam :: (a -> Expr b) -> Expr (a -> b)
  ExprApp :: Expr (a -> b) -> Expr a -> Expr b

eval :: Expr a -> a
eval (ExprBool b) = b
eval (ExprLam f) = \x -> eval (f x)
eval (ExprApp f x) = (eval f) (eval x)
```

Figure 1.7: Generalized algebraic datatype of typed expressions and evaluation function.

Context

GADTs have deep roots in type theory and functional programming research, where they were preceded by the more general concept of *inductive families* in dependently-typed languages. Inductive families allow types to be indexed by values, and in dependently-typed settings, even by other types. For example, Dybjer [39] gave inductive definitions of lists indexed by length and lambda terms indexed by the number of free variables. Such examples can also be encoded using GADTs and type-level natural numbers (*e.g.*, as in Figure 1.6), avoiding full inductive families in the language.

The idea of using only types as indices to be able to use more practically tangible type-checking algorithms can be traced back to an unpublished note by Augustsson and Petersson [10]. Building on type families in ALF, the authors studied an extension of Hindley-Milner type checking that permits type parameters in result types to vary. They illustrated its usefulness by encoding typed grammars for parsing and an expression language in which only well-typed terms can be constructed, similar to Figure 1.7. Their "silly type families" anticipated what later came to be called GADTs, preceding the term by years. Meanwhile, in the broader programming languages community of the late 1990s, there was growing interest in bringing lightweight forms of dependent types into real-world functional languages.

One precursor for GADTs was research of *phantom types* in Haskell. A phantom type is a type parameter with no corresponding runtime value, used solely as a compile-time tag. By designing APIs around phantom types, programmers could enforce invariants through the type system. For instance, in Figure 1.6, the type `S Z` carries a phantom parameter `Z` with no value attached, but can be used to represent 1 on the type level. A notable advance was the notion of *first-class phantom types* proposed by Cheney and Hinze [29], which allowed to specify type-level equalities for phantom parameters and allowed programmers to express type equalities as constraints, providing stronger type guarantees.

At the same time, Xi et al. [144] introduced a similar idea of *guarded recursive datatype constructors* in ML, via the calculus $\lambda_{2,G\mu}$, which incorporated type equalities directly into type definitions. The authors demonstrated how a version of ML extended with these features ($\text{ML}_{2,G\mu}$) could be elaborated into their calculus.

These two lines of work (first-class phantom types and guarded recursive datatypes) were essentially equivalent in expressive power, and together foreshadowed GADTs, albeit in different languages. Both generalized ADTs by allowing constructor return types to fix type parameters to specific types, thereby enabling the type system to reason about type indices. Because constructors could "pin down" type parameters, both approaches allowed to enforce constraints among constructor arguments and the resulting type.

By the early 2000s, the idea that types could carry richer information via indices or phantom parameters was established, though either confined to dependent type theory or encoded in an ad-hoc manner in Haskell/ML-languages. Importantly, the early work on GADTs also provided solid type-theoretic foundations: both Cheney and

Hinze [29] and Xi et al. [144] proved syntactic soundness of their systems, ensuring that well-typed programs could not "go wrong" at runtime. From the operational semantics point of view, their versions of GADTs behaved much like ADTs, but with stronger static guarantees.

The next challenge was integration into practical programming languages. It was quickly realized that type inference for GADTs is tricky: in fact, it was shown that without some restrictions or annotations, type inference for GADTs is undecidable in general. However, with additional annotations, type inference can be made decidable, as shown by Peyton Jones et al. [95]. The design guaranteed that if a GADT-using program has relatively small number of type annotations, then a relatively simple algorithm can infer its types. In practice, GHC adopted a strategy in which functions using GADT pattern-matching often require a type signature to guide inference, reflecting these complexities of working with GADTs.

Concrete implementations soon followed. Sheard [110] pioneered an implementation of GADTs in the experimental language Ωmega, which was the first implementation of Generalized Algebraic Datatypes in a programming language that does not have dependent types. GHC added GADTs as an extension around 2005, after which other languages followed. Notably, OCaml incorporated GADTs in version 4.00 (2012), with the type-theoretic integration described by Garrigue and Rémy [50].

Research Questions

Although GADTs were extensively studied from the implementation and syntactic point of view, their semantic models were only investigated from the categorical model point of view. In this work, we pluck the fruit and study the semantics of GADTs. We introduce a calculus that incorporates the notion of type equality and show how it supports writing programs with GADTs. To express GADTs and reason about pattern-matching on them, we extend the calculus with provability judgments for type equalities, divided into two classes of rules. The first class, *discriminability rules*, allows us to eliminate impossible cases. The second class, *injectivity rules*, provides a way to derive equalities of subcomponents of compound types.

Injectivity of type constructors. Injectivity of type constructors presents challenges for semantic studies of programming languages. In particular, it is well-known that injectivity of type constructors does prevent one from providing set-based semantical models. For example, in a simply typed calculus with empty and product types, interpreting types as sets of valid values of these types makes $[\![A \times \mathsf{void}]\!] = [\![B \times \mathsf{void}]\!]$ trivially true, since both types are realized with empty sets. Injectivity would then force $[\![A]\!] = [\![B]\!]$, which is not true for arbitrary A and B. With impredicative polymorphism combined with injectivity, even syntactic models exhibit unexpected behavior. Admitting injectivity for impredicative types allows one to derive false[4]. In this work,

[4] In fact, injectivity of any higher-kinded type constructor allows one to derive absurdity: https://github.com/idris-lang/Idris-dev/issues/3687.

we construct a model that supports injectivity of type constructors using a novel combination of normalization by evaluation and logical relations. The key feature of our model is that it preserves syntactic information about type equalities, but does not impede relational reasoning.

Semantics. Earlier work has sometimes established syntactic soundness results for calculi with GADTs, but the semantics of GADTs has been more extensively studied in the setting of *functoriality*. In particular, Johann *et al*. [65–70] developed categorical models of languages where type constructors are functorial, exploring whether GADTs can be represented as initial algebras of strictly positive functors, similar to the approach used more generally for inductive types. That is, instead of studying a concrete language that has the capability to express GADTs, the authors studied conditions on functors that can be used to represent GADTs. In these works, the authors posed the question if nested types and GADTs are syntactic generalizations of ADTs; is it possible to generalize the functorial treatment of ADTs to GADTs? Johann and Ghiorzi [67] answered this question positively for nested types, where the authors construct a parametric model that allows one to express all nested types appearing in the literature and provide parametricity results. For GADTs, however, Johann et al. [70] showed that parametric reasoning rules out their characterization as initial algebras.

In contrast to their work, we focus on semantics of one concrete language, inspired by real-world languages, that supports impredicative polymorphism, general recursive types, and internalized type equalities, with no restriction on what type constructors we allow, which makes our approach to semantics of GADTs different. The authors target diverse host languages, while our treatment of GADTs relies on injectivity and discriminability of type constructors in a particular language. In addition to that, in our work, we not only target parametricity results but also focus on *representation independence*. That is, we construct a model that supports showing contextual equivalence of programs that utilize GADTs and implement the same interface. In addition to that, our work is fully mechanized in Rocq, allowing for mechanized reasoning about programs using GADTs.

Another line of work, explored by Vytiniotis and Weirich [137], studies a version of System F_ω extended with a single GADT: the equality type. Although this approach also pursues relational reasoning, the authors provide only congruence rules for internalized type equalities, and it remains an open question whether their system can represent programs that make essential use of GADTs. In particular, because their language is terminating, it seems unlikely that injectivity rules are admissible in their model.

Results

We develop a semantic model for a version of System F_ω extended with built-in type equalities, sufficient to express GADTs as they appear in practical programming languages such as Haskell and OCaml. For this calculus, we give the first unary and

binary models, based on a novel combination of normalization by evaluation and logical relations. Using these models, we obtain the following results:

- We show that combining impredicative polymorphism with injectivity of polymorphic types yields a new encoding of non-terminating computation. Consequently, the language is not strongly normalizing, even in the absence of effects or recursion, which shows that it cannot be encoded in System F_ω;
- We demonstrate that the unary model can be used to reason semantically about programs with GADTs, by analyzing programs that are semantically valid but syntactically ill-typed, and establishing their properties;
- Using the binary model, we establish that parametric reasoning is possible even in the presence of syntactic constraints on types. In particular, we prove contextual equivalence between two programs, one of which is implemented using GADTs.
- The only limitation of our approach is that relational reasoning is restricted to types of the base kind.

1.4.2 Context-Dependent Effects

In this section, we provide background and an overview of context-dependent effects, their role in modern programming languages, and the challenges of describing their semantics. We focus in particular on languages with undelimited continuations (such as those featuring `call/cc` and `throw`) and on languages that support a more refined family of continuation primitives, *delimited continuations*. For concreteness, we use OCaml-like syntax, and adopt `call/cc` and `throw` as representative operators for presenting context-dependent effects.

Intuitively, context-dependent effects provide an interface to "the rest of the computation." While general continuation operators are not widely available in mainstream languages, one prominent instance of this class of effects, exceptions, is ubiquitous. For example, consider the program in Figure 1.8, which implements an early return from a subroutine using exceptions. The program defines a new exception and then proceeds to define a function `find_first_even` that traverses a list of integers, raising the exception with the first even number it encounters. The function `main` calls `find_first_even` on the list `[1; 2; 3; 4]`. If no exception is raised, the program reports that no even numbers were found. However, in this case `find_first_even` raises an exception, which aborts the remainder of the `try` branch (skipping `Printf.printf "No even numbers found!\n"`) and transfers control to the exception handler under the keyword `with`. The handler then reports the first even number found. In this example, the enclosing `try ... with ...` acts as a delimiter, separating the notion of "the rest of the computation" into two parts: the continuation between the exception-raising code and the handler, and the remainder of the program. The operator `raise (Found x)` discards the intermediate

```
exception Found of int

let find_first_even (xs: int list): unit =
  List.iter (fun x ->
    if x mod 2 = 0 then raise (Found x)
  ) xs

let main () =
  let xs = [1; 2; 3; 4] in
  try
    find_first_even xs;
    Printf.printf "No even numbers found!\n"
  with
  | Found x ->
    Printf.printf "Answer: %d\n" x
```

Figure 1.8: Early return with exceptions.

continuation, effectively forgetting part of the computation. Crucially, the place where `raise (Found x)` was invoked can change the semantics of the program.

Although exceptions are ubiquitous in programming, they do not fully illustrate the expressive power of context-dependent effects. The example in Figure 1.8 shows how to abort part of a computation and handle a raised exception, but it does not treat "the rest of the computation" (or *continuation*) as a first-class object, delegating it to the exception handling mechanism. In Chapter 3, we study two forms of continuation-based effects: *undelimited* continuations, where the continuation represents the entire remainder of the program, and *delimited* continuations, where the programmer can specify a slice of the rest of the computation.

To demonstrate context-dependent effects in their more general form, we use examples in a hypothetical extension of OCaml, describing their intended meaning informally. The interface shown in Figure 1.9 is based on an extension of Standard ML[5], supporting the operators `call/cc` and `throw` as in Duba et al. [37]. These operators are canonical examples of undelimited continuations.

Here, `type 'a cont` internalizes the notion of a continuation that expects a value of type `a`. The operator `throw` resumes a continuation by supplying it with such a value, while also aborting the current continuation. The second operator, `callcc` (standing for "call-with-current-continuation"), captures the current continuation and binds it to a variable, allowing the user to use it later. In this example, we assume the continuation primitives are *multi-shot*, meaning the same continuation can be invoked multiple times.

[5]This addition to Standard ML with impredicative polymorphism made the language unsound, which illustrates the complexity of using continuations. The issue was raised in `sml` mailing list:

```
type 'a cont
val throw: 'a cont -> 'a -> 'b
val callcc: (('a cont) -> 'a) -> 'a
```

Figure 1.9: Hypothetical API for `call/cc` in OCaml.

```
let find_first_even_cc k (xs: int list): unit =
  List.iter (fun x ->
    if x mod 2 = 0 then
      throw k (Printf.sprintf "Answer: %d\n" x)
  ) xs

let main () =
  let xs = [1; 2; 3; 4] in
  let s =
    callcc (fun k -> find_first_even k xs;
                     "No even numbers found!\n")
    in
  Printf.printf s
```

Figure 1.10: Early return with `call/cc`.

This interface also allows users to implement functionality similar to exceptions. For example, consider the program in Figure 1.10. Essentially, it implements the same early return functionality as the program in Figure 1.8, but this time using continuation primitives.

The function `main` defines the same list, but computes the output string in a different way. It first captures the current continuation with `callcc` (in this case, the continuation is `let s = _ in Printf.printf s`, where the underscore indicates the value it expects) and passes this continuation to `find_first_even_cc`. If `find_first_even_cc` does not invoke the continuation, then `s` is instantiated with `"No even numbers found!\n"`, and the program terminates normally, displaying this string. However, unlike the exception-based version, `find_first_even_cc` now takes the continuation `k` as an explicit argument. If `find_first_even_cc` encounters an even number, it uses `k` to abort the remainder of the computation and return the first even number directly to the point where `callcc` was invoked.

In addition to early return, context-dependent effects can be used for a plethora of other use cases, *e.g.*, cooperative concurrency, backtracking, generators. In the following, we provide a slightly more involved example of their expressivity. As an illustration of their expressivity, consider the program in Figure 1.11, which implements a primitive form of backtracking. The program enumerates solutions to

https://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html.

the simple constraint $a * b = 2$ using a single mutable cell to store the current failure handler.

Intuitively, the function `choose` introduces a choice point. It immediately returns its first alternative while also installing a new failure handler in `cell`. When invoked, this handler restores the previous failure handler and resumes the continuation of `choose` as if it had instead returned the second alternative.

Let us step through the execution of `solve ()`:

- `a = choose 0 1`: installs a new failure handler `h1 = fun () -> cell := fail; throw k 1`, and returns `0`. Here, `k` captures both the subsequent choice of `b` and the test.
- `b = choose 0 2`: installs another handler `h2 = fun () -> cell := h1; throw k 2`, and returns `0`. Now, `k` contains only the test code.
- `if a * b = 2 then (a, b) else !cell ()`: the test fails, so we invoke the current failure handler.
- `!cell ()`: restores `h1`, tries again with `b = 2`, but the test still fails, triggering another call to the failure handler.
- `!cell ()`: reverts to the initial handler, setting `a = 1`. The continuation now executes both the choice of `b` and the test.
- We repeat the process with `b = 0`, fail, and finally reach the successful pair `a = 1, b = 2`.

The trace is summarized below:

$$(0,0) \to \texttt{fail} \to (0,2) \to \texttt{fail} \to (1,0) \to \texttt{fail} \to (1,2) \to \texttt{success}$$

This example illustrates both the expressive power and the complexity of context-dependent effects. They enable the encoding of powerful programming patterns, such as backtracking, but at the cost of making program reasoning nontrivial. In particular, reasoning becomes less local: the behavior of a programming language construct may be different depending on the surrounding context. For instance, in Figure 1.11, understanding the outcome of a `throw` requires knowing where `callcc` was invoked. In the next section, we review the historical development and current state of semantic studies of context-dependent effects.

Context

Context-dependent effects have a long history in programming languages. Early studies were motivated by the connection between imperative constructs in ALGOL 60 and extensions of the lambda calculus. One of the first context-dependent effects was introduced by Landin [81], who proposed the J operator in ISWIM. This operator

```
exception Fail
let fail: unit -> 'a = fun () -> raise Fail
let cell: (unit -> 'a) ref = ref fail

let choose (x: 'a) (y: 'a): 'a =
  callcc (fun k ->
    let old = !cell in
    cell := (fun () -> cell := old; throw k y);
    x
  )

let solve () =
  let a = choose 0 1 in
  let b = choose 0 2 in
  if a * b = 2 then (a, b) else !cell ()

let main () =
  try
    let (a, b) = solve () in
    Printf.printf "Solution: (%d, %d)\n" a b
  with Fail ->
    Printf.printf "No solution!\n"
```

Figure 1.11: Backtracking with `call/cc`.

captured the current continuation as a first-class value, providing a way to translate ALGOL non-local jumps and labels into closures in ISWIM [82]. Similarly, Reynolds [104] divided ALGOL 60's imperative features into two classes: state operations and control instructions. To study the latter, he introduced the `escape` operator, representing jumps. The J operator and the `escape` operators anticipated `call/cc`, later formalized by Felleisen [41], although its semantics was originally given only operationally via the SECD machine. Control operators first appeared in real-world functional programming languages in Scheme [128], which introduced `catch/throw` for non-local escapes (shown to be equivalent to `escape`), and later added `call/cc`.

Delimited continuations. The `call/cc` effect we discussed is an example of an undelimited continuation manipulation: it captures the entire rest of the computation. Because reasoning about undelimited continuations is difficult, finer-grained operators were proposed in the form of *delimited continuations*. In comparison to undelimited continuations, which capture the rest of the program, delimited continuations allow programmers to explicitly control the extent of captured continuations. In case of delimited continuations, in addition to the capture operation (which allows access to continuations), and invocation operation (which allows to use continuations), language

is supposed to expose an operations to delimit control. Various forms of delimited continuations appeared in Felleisen [42], Sitaram and Felleisen [116], with the modern `shift`/`reset` operators introduced by Danvy and Filinski [33] together with a novel type system [32]. Later, a comprehensive taxonomy of delimited continuations was given by Dyvbig et al. [40].

Denotational semantics of context-dependent effects. The study of non-local jumps in imperative languages led to continuation-passing style (CPS) semantics, pioneered by Strachey and Wadsworth [126], who modeled an ALGOL-like language with `goto`. In CPS, the denotational semantics carries the continuation explicitly as an additional parameter, making it well-suited for context-dependent effects. In particular, the CPS interpretation of `throw` from our previous example would discard the current continuation, and the interpretation of `callcc` would use the current continuations in the body of `callcc`. That is, the interpretation effectively does two things at the same time: it performs CPS conversion of programs, and computes denotations of CPS-converted programs. This approach was later extended to functional languages with undelimited continuations [45], and then to delimited continuations [33]. Dyvbig et al. [40] generalized these results by presenting a monadic framework for various kinds of delimited continuations in CPS, showing that this approach suffices for all types of existing flavors of delimited continuations. In parallel, Sitaram and Felleisen [117] introduced an alternative class of direct denotational models for control operators, with the goal of establishing full abstraction.

(Guarded) Interaction Trees. Denotational semantics provides a powerful framework for studying programming languages, but constructing suitable models is often challenging and highly language-specific. Mechanization in proof assistants adds further difficulty, particularly when working with domains. One might argue that the domain-theoretic model of the untyped lambda calculus [107, 108], being a denotational semantics for a Turing-complete language, could in principle serve as a universal model for all languages. In practice, however, encoding diverse effects within this model is often not straightforward, and the resulting semantics becomes harder to reason about, also making practical mechanization more complicated. Interaction Trees (ITrees), introduced by Xia, Zakowski, He, Hur, Malecha, Pierce, and Zdancewic [145], were proposed as a modular solution to this problem: they provide a mechanized domain for expressing the denotational semantics of first-order languages, supporting divergence and first-order effects (*e.g.*, I/O, first-order state). That is, ITrees act as one particular mechanized domain that can be applied to a plethora of languages, and comes equipped with tools for executing them as programs and equational reasoning about weak bisimilarity of ITrees.

However, ITrees are best suited to first-order effects: higher-order functions must be encoded (for instance, Vistrup et al. [136] used closures, when giving semantics of lambda-functions, for function application the authors perform syntactic substitution), which complicates reasoning about functional languages like Haskell or OCaml.

$$
\begin{aligned}
&\mathsf{coinductive\ type}\ \mathbf{IT}_F(A) \triangleq \\
&\quad \mid \mathsf{Ret} : A \to \mathbf{IT}_F(A) \\
&\quad \mid \mathsf{Tau} : \mathbf{IT}_F(A) \to \mathbf{IT}_F(A) \\
&\quad \mid \mathsf{Vis} : \prod_{B \in Type} \big(F(B) \times (B \to \mathbf{IT}_F(A))\big) \to \mathbf{IT}_F(A)
\end{aligned}
$$

(a) Type of ITrees.

$$
\begin{aligned}
&\mathsf{guarded\ type}\ \mathbf{IT}_E(A) \triangleq \\
&\quad \mid \mathsf{Ret} : A \to \mathbf{IT}_E(A) \\
&\quad \mid \mathsf{Fun} : \blacktriangleright(\mathbf{IT}_E(A) \to \mathbf{IT}_E(A)) \to \mathbf{IT}_E(A) \\
&\quad \mid \mathsf{Err} : \mathtt{Error} \to \mathbf{IT}_E(A) \\
&\quad \mid \mathsf{Tau} : \blacktriangleright\mathbf{IT}_E(A) \to \mathbf{IT}_E(A) \\
&\quad \mid \mathsf{Vis} : \prod_{\mathtt{i} \in \mathtt{I}} \big(Ins_{\mathtt{i}}(\blacktriangleright\mathbf{IT}_E(A)) \times (Outs_{\mathtt{i}}(\blacktriangleright\mathbf{IT}_E(A)) \to \blacktriangleright\mathbf{IT}_E(A))\big) \to \mathbf{IT}_E(A)
\end{aligned}
$$

(b) Type of GITrees.

Figure 1.12: Types of (G)ITrees.

Guarded Interaction Trees (GITrees), introduced by Frumin, Timany, and Birkedal [49], provide an alternative to ITrees to handle higher-order functions and effects. They provide denotational semantics and reasoning principles for languages employing effects like I/O and higher-order state. That is, GITrees is also one particular domain for expressing denotational semantics, albeit tinkered to work with a different class of languages (featuring higher-order effects and higher-order functions). Unlike ITrees, GITrees do not rely on bisimilarity but instead use a program logic. More concretely, GITrees come equipped with reduction relation, which specifies a way to compute using user-supplied rules for effects. On top of this reduction relation, it is possible to define *weakest precondition*, which specifies the conditions required for a safe execution of a GITree, and, moreover, specifies a postcondition if the GITree reduces to a function or a primitive value. The program logic for GITrees is implemented on top of the Iris framework, and we refer the reader to Jung et al. [73] and [49, Section 6] for details.

While closely related, ITrees and GITrees are defined in different settings. ITrees are defined as a *coinductive* type, whereas GITrees are defined as a *guarded*[6] type. We show both definitions in Figure 1.12 and briefly highlight their differences; for a more detailed account of GITrees we refer the reader to [49, Section 3] and to [145, Section 2].

Both ITrees and GITrees are parameterized by a type A of return values, with the

[6]Note that GITrees are implemented in Rocq, which does not support guarded types natively. Instead, this type is implemented on top of an implementation of guarded type theory in Rocq — Iris framework. Here, to simplify the presentation, we use *Type* for both guarded and Rocq types.

constructor Ret producing value elements. Each includes a Tau constructor for representing "silent" steps, which can be used to encode non-terminating computations: via corecursion in the case of ITrees, or via guarded recursion for GITrees.

To represent effects, both types are parameterized by an effect signature, though in different forms. For ITrees, the parameter is a function $F : \mathit{Type} \to \mathit{Type}$, which specifies the type of inputs expected by the used effects. For GITrees, the effect signature is given by a triple $E = (\mathtt{I}, \mathit{Ins}_-, \mathit{Outs}_-)$, where I is a set of operation symbols (available effects) and $\mathit{Ins}, \mathit{Outs} : \mathtt{I} \to (\mathit{Type} \to \mathit{Type})$ are functions specifying, respectively, the input and output types of each operation. Note that both *Ins* and *Outs* are parameterized by GITrees themselves. While introducing negative occurrences, and thus preventing from defining this type as a coinductive in Rocq, this parameterization allows to express higher-order effects, taking and returning GITrees.

In addition, GITrees introduce two constructors not present in ITrees. The constructor Fun allows functions themselves to be represented as values inside the type of GITrees. Because this involves negative occurrences, it cannot be added to a coinductive definition, similar to the higher-order version of Vis. Finally, GITrees include an Err constructor to account for runtime errors, such as attempting to apply a non-function value as a function.

In summary, both frameworks provide mechanized domains for denotational semantics of languages. However, they target different families of languages, with Interaction Trees targeting first-order languages with first-order effects, and offering reasoning using weak bisimilarity. In contrast, Guarded Interaction Trees offer a domain for languages with higher-order functions and effects, while using a different approach to reasoning via program logic.

Research Questions

In this work, we study how to model context-dependent effects within the GITrees framework. Specifically, we extend GITrees with facilities to represent context-dependent effects, and evaluate the framework on canonical examples such as `call/cc` and `shift/reset`.

Context-dependent effects in Guarded Interaction Trees. While Frumin, Timany, and Birkedal [49] developed GITrees to support higher-order functions and effects, they left context-dependent effects as future work. We fill this gap by adapting the framework to represent and reason about such effects.

Direct semantics of context-dependent effects. Previous work [33, 40, 45] provided CPS models for context-dependent effects, but CPS reasoning has drawbacks. In particular, relating CPS semantics of a full language with direct semantics of a fragment lacking continuation operators can be difficult. By contrast, our approach models both undelimited and delimited continuations *directly*, so that adding continuation effects does not force the adoption of CPS for the rest of the language.

An alternative line of work is *extensible denotational semantics* [28], which represents effects as resources managed by an administrator. This framework supports extension by introducing new resources and has been used to build a fully abstract direct semantics for context-dependent effects [117]. Our work differs in that we extend GITrees rather than using classical domain theory. This makes it possible to mechanize results in a proof assistant and to leverage separation logic for reasoning about effect resources. In addition to that, our work is fully mechanized.

Results

We extend Guarded Interaction Trees with primitives for effects that observe continuations, thereby enabling the representation of context-dependent effects. Our contributions are as follows:

- We provide a modification to the GITrees framework, allowing one to specify context-dependent effects;
- We extend the GITrees program logic with general rules for context-dependent effects, and derive specialized rules for concrete operators such as `call/cc` and `shift/reset`;
- We give sound and adequate direct denotational semantics for a language with `call/cc` in the GITrees framework;
- We give sound and adequate direct denotational semantics for a language with `shift/reset`;
- We demonstrate interaction between two languages: one with `shift/reset` and one with higher-order state;
- We show that our extension is conservative: previous examples remain valid with the extended framework.

1.4.3 Recursive Domain Equations

In this section, we motivate the mechanization of guarded recursive domain equations and compare our approach with existing mechanizations.

Recursively defined objects are ubiquitous in the semantics of programming languages. A standard way to study them is through recursive domain equations, defined as suitable functors over suitable categories. That is, given an endofunctor $F : \mathcal{C} \to \mathcal{C}$, the goal is to find an object X such that $F(X) \simeq X$. Since Scott's seminal report on domain theory [107], many categorical frameworks have been developed to solve such equations, ranging from concrete categories, such as the category of chain complete partial orders and the category of metric spaces, to more abstract settings based on certain classes of *enriched categories*. One line of work connects recursive domain equations to step-indexing techniques [24, 73], which have been

applied to define denotational semantics [93] and expressive logics for reasoning about programs [73]. In particular, this approach is used to define the type of GITrees, the domain of interpretation used in Chapter 3 and briefly discussed in Section 1.4.2.

Context

Although step-indexing techniques were successfully applied in these works, most implementations in proof assistants were based on step-indexing over the natural numbers. While sufficient for many applications, this setting breaks down when one requires the so-called *existential property*[7], which forces indexing over larger ordinals. This motivates the need for *transfinite step-indexing*. Two main approaches have been explored: working in categories of (pre)sheaves, and working with *(complete) ordered families of equivalences* ((C)OFE, with COFE being a subcategory of OFE) [52].

In the sheaf-based direction, Birkedal, Møgelberg, Schwinghammer, and Støvring [24] developed *synthetic guarded domain theory* inside the category of sheaves over complete Heyting algebras with a well-founded basis. This provided a categorical framework for guarded recursion, generalizing results beyond step-indexing over ω.

Another approach is to use *(complete) ordered families of equivalences* ((C)OFEs), which can be understood as a particular formulation of (complete) ultrametric spaces that is well-suited for mechanization. Step-indexed (C)OFEs over the natural numbers were first employed for solving recursive domain equations in the ModuRes library [112], and were later developed into the more general Iris framework [73]. An alternative is Transfinite Iris [119], which extends the Iris, generalizing Iris with transfinite step-indexing. By replacing finite step-indices with transfinite ones, the authors validated the existential property and enabled reasoning about liveness properties such as termination and termination-preserving refinement in higher-order stateful programs. In doing so, they also solved a novel recursive domain equation for transfinitely step-indexed propositions and fully mechanized their results in Rocq.

Research Questions

Our work addresses two questions.

First, we seek to simplify the original categorical constructions for solving guarded recursive equations. Compared to Birkedal et al. [24], who work in the general setting of sheaves over complete Heyting algebras with well-founded bases, we work in the specialized case of presheaves over the ordinal category. While this makes our framework less general, it has two key advantages: it avoids the technical overhead of sheaves, and it is significantly easier to mechanize. In particular, working with sheaves in proof assistants is technically intricate. In contrast with sheaves, presheaves offer a simpler and more direct setup for mechanization in proof assistants while remaining equivalent to the sheaf-based approach in the case of ordinals. Using this simplification, we mechanize our solution in Rocq.

[7] We explain this property in Appendix A.1.

Second, we provide a more general solver for recursive domain equations than the one available in Transfinite Iris [119]. Their solver works only for equations of the shape $F\colon \mathrm{OFE} \times \mathrm{OFE}^{\mathsf{op}} \to \mathrm{COFE}$, which suffices for Iris propositions but excludes other important cases. Importantly, equations of this shape differ from those supported by Iris: $F\colon \mathrm{COFE} \times \mathrm{COFE}^{\mathsf{op}} \to \mathrm{COFE}$. Since COFE is a subcategory of OFE, this restricts the set of equations that can be solved. For example, this restriction prevents solving the recursive domain equation for GITrees [49, Section 3]. In contrast, our approach does not impose such restriction: we solve guarded equations of the more general form $F\colon \mathrm{Pr}(\mathbf{Ord}) \times \mathrm{Pr}(\mathbf{Ord})^{\mathsf{op}} \to \mathrm{Pr}(\mathbf{Ord})$, where $\mathrm{Pr}(\mathbf{Ord})$ denotes presheaves over the ordinal category.

Results

We develop a systematic account of solving guarded recursive domain equations directly in the category of presheaves over ordinals. Our contributions are twofold:

- We construct solutions to guarded mixed-variance domain equations in the simplified presheaf setting;
- We provide a complete mechanization of the theory in Rocq, avoiding the technical complications of sheaves while retaining unrestricted solver for RDEs in the ordinal case.

This yields a framework that is both mathematically simpler and more suitable for mechanized reasoning.

1.5 Future Work

In this section we outline several problems related to this dissertation that remain open for future research.

Relating GITrees. Although GITrees provide both an equational theory and a program logic on top of their reduction semantics, relating two GITrees remains a significant challenge. Such relations are crucial for applications such as compiler verification, where the source and target languages are both given denotational semantics in terms of GITrees, and one wishes to connect them via weak bisimilarity or a logical relation. Our preliminary survey showed that adequacy of either weak bisimilarity or expressive binary logical relations requires the *existential property*, which is not currently available in the Iris-based mechanization of GITrees.

Mechanizing GITrees with Transfinite Guarded Type Theory. Extending the mechanization of GITrees to support relational reasoning therefore requires access to the existential property. Transfinite Iris already provides this property, but in its current form does not support the definition of GITrees. Determining whether Transfinite Iris

can be generalized to define GITrees is an open question and could offer a resolution to this problem. An alternative direction is to further develop the framework introduced in Section 1.4.3 and further discussed in Chapter 4, which supports both the existential property and the definition of GITrees. However, this framework still requires further development before it can serve as a practical foundation for large-scale mechanized semantics. In particular, in comparison to (Transfinite) Iris, our framework lacks a proof mode [79], which would considerably simplify reasoning.

1.6 Statement of Personal Contributions

This dissertation contains text and material from three publications that I contributed to during my Ph.D. Publications are included with only minor formatting changes. Publications and my contributions are listed below:

- The Essence of Generalized Algebraic Data Types, Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, Lars Birkedal, POPL 2024 [114].
  I contributed to the definition of the semantic model of the language. I also contributed to the writing of the unary and binary models sections in the paper. This publication was also used for my qualifying examination progress report. The manuscript is included in Chapter 2.

- Context-Dependent Effects in Guarded Interaction Trees, Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, Lars Birkedal, ESOP 2025 [122].
  In this project, I contributed to the modification of the original definition of reification for GITrees to allow them to capture continuation, allowing one to express control-dependent effects. To show that our approach is viable, I contributed to providing a sound and adequate direct interpretation of a language with `call/cc` and a language with `shift/reset`. I contributed to all parts of the paper and led the research on this project. The manuscript is included in Chapter 3.

- Solving Guarded Domain Equations in Presheaves over Ordinals and Mechanizing It, Sergei Stepanenko, Amin Timany, FSCD 2025 [120].
  In this project, I focused on the mechanization of the recursive domain equation solver. The manuscript is included in Chapter 4.

In addition, all materials in this thesis are formalized in the Rocq Prover, with artifacts publicly available:

- The Essence of Generalized Algebraic Data Types [113];
- Context-Dependent Effects in Guarded Interaction Trees [123];
- The Rocq Mechanization of Solving Guarded Domain Equations in Presheaves Over Ordinals [121].

Part II

Publications

CHAPTER 2

The Essence of Generalized Algebraic Data Types

Abstract

This paper considers direct encodings of generalized algebraic data types (GADTs) in a minimal suitable lambda-calculus. To this end, we develop an extension of System F_ω with recursive types and internalized type equalities with injective constant type constructors. We show how GADTs and associated pattern-matching constructs can be directly expressed in the calculus, thus showing that it may be treated as a highly idealized modern functional programming language. We prove that the internalized type equalities in conjunction with injectivity rules increase the expressive power of the calculus by establishing a non-macro-expressibility result in F_ω, and prove the system type-sound via a syntactic argument. Finally, we build two relational models of our calculus: a simple, unary model that illustrates a novel, two-stage interpretation technique, necessary to account for the equational constraints; and a more sophisticated, binary model that relaxes the construction to allow, for the first time, formal reasoning about data-abstraction in a calculus equipped with GADTs.

2.1 Introduction

Since their introduction twenty years ago, *generalized algebraic data types* [29, 111, 144], commonly referred to as GADTs, have become a staple of modern functional programming languages. First introduced as an extension of the Glasgow Haskell Compiler, they were since adopted by OCaml, Scala and many other programming languages, due to their ability to establish more precise specifications for algebraic type constructors. These implementations, and the attendant difficulties, have also resulted in a rich literature that tackles the problems of type inference in languages with ML-style polymorphism and GADTs [71, 103]. However, one feature that remains understudied is the semantics of GADTs and the nature and expressivity of these types. Since the seminal paper of Xi et al. [2003], GADT calculi have

usually been equipped with primitive notions of algebraic types and their constructors. While this is desirable for studying the problems of inference, it tends to obscure the essence of the structure of the types, and where their expressive power stems from. In this paper, we intend to shed more light on these aspects of GADTs by introducing what we believe to be the first calculus that does not rely on the notion of a data type constructor to express GADTs. We then proceed by studying soundness and expressive power of the calculus, as well as provide some interesting and difficult open problems.

Beyond algebraic data types. Algebraic data types are one of the fundamental features of modern functional programming languages, arguably as important to programming practice as higher-order functions themselves. One of the canonical examples is that of a binary tree with nodes labeled by elements of some type, which can be defined as follows, using Haskell syntax:[1]

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

These types are often modeled in λ-calculi with a combination of simple types — disjoint sums for distinct constructors, products for their arguments — and iso-recursive types [96, Chapter 21]. For instance, the type of trees labeled with elements of type α would correspond to the following type:

$$\mathsf{Tree}\ \alpha \triangleq \mu\beta.\ \mathsf{unit} + (\beta \times \alpha \times \beta)$$

Above, the variable β bound by μ stands for the recursive occurrences of the Tree type constructor.

There are two important points to note about the type of trees. First, it really is a *type constructor*, rather than a type: we only get a type after the definition of Tree is applied to an argument, the type of labels. Second, the definition is *uniform* in the argument: all the recursive arguments are indexed with the same type of elements as the complete tree. Over the last quarter of a century, numerous generalizations have been proposed to extend the expressive power of algebraic data types and allow definitions to more precisely describe the shape of the data.

The most natural way to relax the constraints on the shapes of algebraic data types is to remove the uniformity condition; types obtained in this way are often dubbed *nested* algebraic data types. A simple example due to Bird and Meertens [20] is the type of perfectly balanced trees, defined below:

```
data PTree a = PLeaf | PNode a (PTree (a * a))
```

This definition models a perfectly balanced tree as a list of levels; at each depth, we require twice as many elements as at the previous one, by virtue of the fact that the recursive call to the type constructor is taken at type `a * a`. Since the type is no

[1] In this paper, we use Haskell syntax for presenting examples. However, our language differs from Haskell, as we use a different evaluation strategy. In this paper, all examples use call-by-value evaluation strategy, *i.e.* before performing any reductions, arguments should be reduced to values.

longer uniform, simple recursive types no longer suffice to encode it in a λ-calculus based system. We may nonetheless encode it in a version of polymorphic λ-calculus with recursive type constructors, which forms the basis of most modern functional programming languages. Under this extension, type-level recursion can be used to define not only ground types but also proper type constructors. This allows us to encode the type of perfectly-balanced trees as follows:

$$\mathsf{PTree} \triangleq \mu\beta :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha :: \mathsf{T}.\ \mathsf{unit} + (\alpha \times (\beta\ (\alpha \times \alpha))).$$

Above we define PTree as a recursively-defined type constructor by specifying that the recursive variable β shall map types to types; then we introduce the type α of labels using a type-level lambda-abstraction. After that, the encoding follows the pattern described above.

While relaxing the uniformity requirement for algebraic data types is likely the most natural generalization, other possibilities abound. Readers familiar with inductive type families, as introduced in the Calculus of Inductive Constructions [91] or found in the implementations of Coq or Agda, may notice that — even with the relaxed condition — the arguments still satisfy the conditions required of *parameters*, rather than the more general *indices*; the difference is that parameters may be constrained non-uniformly in recursive calls, whereas indices may additionally be non-uniform in the return types of the data constructors.[2] This additional non-uniformity is precisely what ***generalized algebraic data types*** (GADTs) allow for, albeit restricted to the non-dependently typed setting. This may be demonstrated by the following intrinsically well-typed representation of lambda terms:

```
data Tm a where
  Lift :: a -> Tm a
  Lam  :: (a -> Tm b) -> Tm (a -> b)
  App  :: Tm (a -> b) -> Tm a -> Tm b
```

There are several things to observe about the representation above. First of all, we re-use the meta-level types as the indices that indicate the type of the given lambda term. Secondly, although the `Lift` constructor is uniformly inhabited at any type for which we have an element, `Lam` is only ever inhabited at indices that are, syntactically, of the arrow type form, and the indices of the arguments and result of the `App` constructor express nontrivial constraints. This has far-reaching consequences for the behavior of pattern-matching: for instance, if we have an object of type `Tm (a * b)`, we can safely disregard the `Lam` constructor. This ability to enforce constraints on the *return type* of the constructor enables specifications of data types to express many invariants, lending them power that before was only found in dependent type theories. It is worth noting that this property of pattern matching in languages with GADTs expresses the *discriminability* of types whose head constructors are different; discriminability laws of this kind are *not* present in standard dependent type

[2] In the context of dependent type theory, non-uniformity together with propositional equality suffice to encode indices. However, in the absence of equality types or constraints they behave parametrically.

theories, which means that the naïve encoding of GADTs as inductive families in (*e.g.*) Coq does not allow the same programs to be written.[3]

With this additional expressive power comes new problems. Since classic algebraic data types can be directly translated to rather conservative extensions of System F, they are very well studied — not just in terms of type safety, but also stronger properties, such as relational parametricity. In fact, it is the parametricity results that allow us to talk about "free theorems" about simple algebraic data types, such as lists or trees. Since nested, or non-uniform data types can be similarly encoded in well-behaved extensions of System F, many of these results also extend to them. However, this line of results comes to a halt when GADTs are concerned: while many calculi have been developed, the main lines of work have focused on the difficult questions of typechecking in the presence of GADTs [38] or the equally difficult matter of functorial initial algebra semantics for GADTs [66], and very few go beyond type safety results. This paper is an attempt to cross that boundary.

However, in order to investigate existence of relational models of GADTs, we need a calculus that is expressive enough to directly encode GADTs and the associated programming patterns, while at the same time remaining as simple as possible. To this end, in Section 2.2 we develop System $\mathrm{F}^{=i}_{\omega\mu}$, which extends System F_ω with recursive types and *internalized equalities*. System $\mathrm{F}^{=i}_{\omega\mu}$ also contains certain discriminability and type-constructor injectivity rules, which are crucial for GADT-style reasoning. In contrast to most of the GADT calculi, our approach allows us to leave out the notions of constructors of algebraic data types and their associated type constructors. We believe that this allows us to better capture the *essence* of how GADTs behave and remove language constructs that are useful in practice, but superfluous in terms of the semantic study of the calculus and its properties.

After having defined the calculus, we proceed, in Section 2.3, to study its relationship to its well-studied restriction, System F_ω. We establish that even in the absence of recursive types and with certain restrictions on injectivity of type constructors, our calculus is not expressible in F_ω, due to the presence of the typing rules that we use to model GADTs. Thus, any potential translation to the better-studied system must be a full-program compilation. This result cuts against the folklore understanding of GADTs as "existentials and type equalities": we show that the encodings depend crucially on how one can reason about type equality.

Since System $\mathrm{F}^{=i}_{\omega\mu}$ is, to the best of our knowledge, not macro-expressible in known systems, in Section 2.4 we establish its type soundness through a non-trivial syntactic argument, involving a normalization process to cater for equality reasoning, and proceed to investigate the existence of relational models of the calculus in Section 2.5. Since the proof of non-expressibility strongly suggests that the usual approach of interpreting types as predicates or relations on values would not allow us to enforce the necessary injectivity rules, we develop a novel unary model construction that

[3] To account for the needed discriminability in the environment of inductive families in systems like Coq and Agda, one must value the index of `Tm` in a custom inductively defined universe rather than in the ambient universe of existing types.

splits the relational interpretation process in two phases. First, we use a normalization-by-evaluation (NbE) interpretation to account for type-level computation (including injectivity of types); then, the closed, ground subset of the NbE interpretations of types is realized as predicates on values. We observe that while this construction is sufficient to construct a model, it does not allow to reason about syntactically ill-typed programs or data abstraction. To remedy this, we propose a simple extension of the technique, and use it to build a binary relational model that allows us to transport data abstraction results into the realm of GADTs.

With the exception of the expressibility result in Section 2.3, all the constructions presented in this paper are formalized in Coq. We believe that some of those developments, in particular the NbE implementation which bakes functoriality into the construction, thus alleviating some of the post-hoc reasoning necessary, are of independent interest. We discuss the techniques used in the formalization in Section 2.6. Finally, we discuss related work in Section 2.7 and conclude in Section 2.8.

Contributions and Outline. In summary, we make the following contributions:

- We present (in Section 2.2) a novel calculus $F^{=i}_{\omega\mu}$ extending System F_ω with (higher-kinded) iso-recursive types and internalized type-equalities satisfying enough injectivity and discriminability laws to model GADTs.
- We prove (in Section 2.3) that System $F^{=i}_{\omega\mu}$ is not macro-expressible in System F_ω, justifying our claim that GADTs require additional expressive power.
- We prove (in Section 2.4) type soundness of System $F^{=i}_{\omega\mu}$, with a methodologically novel use of normalization-by-evaluation within a syntactic type soundness proof.
- We present (in Section 2.5) two semantic models of System $F^{=i}_{\omega\mu}$, introducing a novel two-stage construction technique that allows us to enforce injectivity of type constructors in a relational model, and reason about syntactically ill-typed programs and data abstraction.
- We have formalized our constructions in the Coq Proof Assistant (Section 2.6).

2.2 Polymorphic λ-calculus with Internalized Type Equalities

In this section we present our core calculus, dubbed System $F^{=i}_{\omega\mu}$, as an extension of System F_ω – the λ-calculus with impredicative polymorphism and type constructors, which is the theoretical foundation of modern functional programming languages. The core feature of our calculus is the *reification* of F_ω type equality, which is a judgment for which there is no *internal* syntax within the F_ω language.

$$
\begin{array}{lll}
\text{kinds} & \kappa & ::= \mathsf{T} \mid \kappa \Rightarrow \kappa \\
\text{constructors} & c & ::= \forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \mid \rightarrow \mid \times \mid + \mid \mathsf{unit} \mid \mathsf{void} \\
\text{constraints} & \chi & ::= \sigma \equiv_\kappa \tau \\
\text{types} & \sigma, \tau & ::= c \mid \alpha \mid \lambda \alpha :: \kappa.\ \tau \mid \sigma\ \tau \mid \chi \rightarrow \tau \mid \chi \times \tau
\end{array}
$$

Figure 2.1: The syntax of kinds, types and constraints.

$$
\forall_\kappa, \exists_\kappa :: (\kappa \Rightarrow \mathsf{T}) \Rightarrow \mathsf{T} \qquad \mu_\kappa :: (\kappa \Rightarrow \kappa) \Rightarrow \kappa \ ^{(\mu)} \qquad \rightarrow, \times, + :: \mathsf{T} \Rightarrow \mathsf{T} \Rightarrow \mathsf{T}
$$

$$
\mathsf{unit}, \mathsf{void} :: \mathsf{T}
$$

$$
\frac{c :: \kappa}{\Delta \vdash c :: \kappa} \qquad \frac{}{\Delta \vdash \alpha :: \Delta(\alpha)} \qquad \frac{\Delta, \alpha :: \kappa \vdash \tau :: \kappa'}{\Delta \vdash \lambda \alpha :: \kappa.\ \tau :: \kappa \Rightarrow \kappa'}
$$

$$
\frac{\Delta \vdash \sigma :: \kappa_1 \Rightarrow \kappa_2 \qquad \Delta \vdash \tau :: \kappa_1}{\Delta \vdash \sigma\ \tau :: \kappa_2} \qquad \frac{\Delta \vdash \chi\ \mathsf{constr} \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \vdash \chi \rightarrow \tau :: \mathsf{T}}
$$

$$
\frac{\Delta \vdash \chi\ \mathsf{constr} \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \vdash \chi \times \tau :: \mathsf{T}} \qquad \frac{(\Delta \vdash \tau_i :: \kappa)_{i \in \{1,2\}}}{\Delta \vdash \tau_1 \equiv_\kappa \tau_2\ \mathsf{constr}}
$$

Figure 2.2: Well-kinded constructors and types, and well-formed equality constraints. The recursive type constructor (marked with a red (μ)) is not considered in a restricted sub-system, $\mathrm{F}^{=i}_{\omega}$.

Syntax of kinds and types The structure of kinds, types and equality constraints is presented in Figure 2.1. The calculus has the standard kind structure: T stands for the kind of proper types, while $\kappa_1 \Rightarrow \kappa_2$ denotes the kind of *type constructors* that produce a type of kind κ_2 given a type of kind κ_1. The types are formed from variables, type abstraction and application, type constants — which include products, disjoint sums, arrows, as well as universal and existential quantifiers, and recursive type constructor — and two types that interact with *type equality constraints*. The first of these is an arrow type predicated on a constraint: its intended meaning is to qualify suspended computations that can only be run if the constraint is valid; the second is an analogue of a product type that provides a value together with a witness of validity of a constraints. The constraints, in turn, are kinded equalities between types. Figure 2.2 provides the kinding rules for types and well-formedness rules for constraints; the two judgments are mutually inductively defined.

By convention, we write binary type constants (*i.e.* arrows, products and sums) infix, and we write $\forall \alpha :: \kappa.\ \tau$ to mean $\forall_\kappa\ (\lambda \alpha :: \kappa.\ \tau)$ — and likewise for existential and recursive types. Note that the arrow and product syntax is somewhat ambiguous;

however, it is always clear from context whether the type is a standard arrow/product, or the constrained variant.

This syntax allows us to express most of the standard types encountered in functional programming: for instance, we can express a uniform algebraic data type $\mathsf{List} \triangleq \lambda\alpha :: \mathsf{T}.\ \mu\beta :: \mathsf{T}.\ \mathsf{unit} + \alpha \times \beta$, with the unit type standing in for the nil constructor, and the pair for the cons. However, the fact that we can form recursive data types at higher kinds allows us to express nested data types, and the constrained types allow us to express GADTs. For example, consider the following type:

```
data Foo :: * -> * where
  C1 :: Bool -> Foo Bool
  C2 :: Foo a
```

We may encode the above by defining $\mathsf{Foo} \triangleq \lambda\alpha :: \mathsf{T}.\ (\mathsf{Bool} \times (\alpha \equiv_{\mathsf{T}} \mathsf{Bool})) + \mathsf{unit}$. We discuss expressivity and examples in more detail in Section 2.2.1.

Discriminability and provability of constraints. With the syntax of kinds and types defined, we can turn to the notion of provability of constraints, which is defined is Figure 2.3. The judgment $\Delta \mid \Phi \Vdash \chi$ denotes that the constraint χ — well-formed in the type-variable context Δ — is provable from the assumptions in Φ, which are also constraints well-formed in Δ. In addition to a rule that allows for the use of assumptions, the rules follow the standard type-equivalence pattern from System F_ω: the type equivalence is a congruence with respect to the type formers, and it is closed under β and η rules. However, in addition to these we also have *injectivity* rules for type constants (including the new variants of arrow and product), marked in purple in the definition. These ensure that if two types formed by the application of the same constructor to the same number of (well-kinded) arguments are equal, then so are, pairwise, their arguments. The consequences of including these rules will be discussed in detail in the following section; the rationale for their inclusion is that this is precisely the reasoning that GADTs require, particularly when performing pattern-matching. For now, let us observe that it is the injectivity rules that allow us to discern, from an assumption $\mathsf{List}\ \alpha \equiv_{\mathsf{T}} \mathsf{List}\ \beta$, that $\alpha \equiv_{\mathsf{T}} \beta$ holds. Note, however, that any essential use of an injectivity rule requires the existence of an assumed equality of two compound types in the context, from which we can derive equalities of components. That can serve as an intuitive explanation for why they are not necessary in System F_ω.

In addition to the provability judgment, we also define a simpler notion of *discriminability* of a constraint. This judgment holds when two types can never be made equal — in principle, whenever the two types are created by distinct type constants. Note that, for the purpose of simplicity, we only consider the top-most constructor, and that the constrained arrow and product types are discriminable from each other, as well as from the types constructed by appropriate application of type constants. The corresponding rules are marked in blue in Fig. 2.3.

Syntax of expressions and values. With the notions of types, constraints and provability defined, we now turn to the term-level of our calculus. The syntax is

$$\frac{c_1 \neq c_2 \qquad (\Delta \vdash c_i\ \overline{\tau}_i :: \kappa)_{i\in\{1,2\}}}{\Delta \Vdash c_1\ \overline{\tau}_1 \#_\kappa c_2\ \overline{\tau}_2} \qquad \frac{\Delta \vdash c\ \overline{\tau} :: \mathsf{T} \qquad \Delta \vdash \chi \to \sigma :: \mathsf{T}}{\Delta \Vdash c\ \overline{\tau} \#_\mathsf{T} \chi \to \sigma}$$

$$\frac{\Delta \vdash c\ \overline{\tau} :: \mathsf{T} \qquad \Delta \vdash \chi \times \sigma :: \mathsf{T}}{\Delta \Vdash c\ \overline{\tau} \#_\mathsf{T} \chi \times \sigma} \qquad \frac{\Delta \vdash \chi_1 \to \tau_1 :: \mathsf{T} \qquad \Delta \vdash \chi_2 \times \tau_2 :: \mathsf{T}}{\Delta \Vdash \chi_1 \to \tau_1 \#_\mathsf{T} \chi_2 \times \tau_2}$$

$$\frac{\Delta \Vdash \tau_2 \#_\kappa \tau_1}{\Delta \Vdash \tau_1 \#_\kappa \tau_2}$$

$$\frac{\chi \in \Phi}{\Delta \mid \Phi \Vdash \chi} \qquad \frac{\Delta, \alpha :: \kappa_a \vdash \sigma :: \kappa_r \qquad \Delta \vdash \tau :: \kappa_a}{\Delta \mid \Phi \Vdash (\lambda \alpha :: \kappa_a.\ \sigma)\ \tau \equiv_{\kappa_r} \sigma[\tau/\alpha]}$$

$$\frac{\Delta \vdash \tau :: \kappa_a \Rightarrow \kappa_r \qquad \alpha \notin \Delta}{\Delta \mid \Phi \Vdash \lambda \alpha :: \kappa_a.\ \tau\ \alpha \equiv_{\kappa_a \Rightarrow \kappa_r} \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \chi \to \sigma \equiv_\mathsf{T} \xi \to \tau}{\Delta \mid \Phi \Vdash \sigma \equiv_\mathsf{T} \tau} \qquad \frac{\Delta \mid \Phi \Vdash \chi \times \sigma \equiv_\mathsf{T} \xi \times \tau}{\Delta \mid \Phi \Vdash \sigma \equiv_\mathsf{T} \tau}$$

$$\frac{c :: (\kappa_i \Rightarrow)_i\ \kappa \qquad \Delta \mid \Phi \Vdash c\ (\sigma_i)_i \equiv_\kappa c\ (\tau_i)_i}{(\Delta \mid \Phi \Vdash \sigma_i \equiv_{\kappa_i} \tau_i)_i}$$

$$\frac{\Delta \mid \Phi \Vdash \chi \times \sigma \equiv_\mathsf{T} \xi \times \tau \qquad \Delta \mid \Phi \Vdash \chi}{\Delta \mid \Phi \Vdash \xi} \qquad \frac{\Delta \mid \Phi \Vdash \chi \to \sigma \equiv_\mathsf{T} \xi \to \tau \qquad \Delta \mid \Phi \Vdash \chi}{\Delta \mid \Phi \Vdash \xi}$$

Figure 2.3: Discriminability of types and provability of equality constraints; for brevity, we omit the rules that make constructor equality a congruent equivalence relation.

defined in Figure 2.4; for simplicity of presentation we use fine-grain call-by-value semantics [86], which significantly reduces the metatheoretical overhead. In larger examples we typically forego these restrictions, assuming a left-to-right, call-by-value reading of the terms.

As a consequence of being an extension of System F_ω, most of the syntax of our calculus is standard, although note that we use $\mathsf{let}\,(*,x) = v\,\mathsf{in}\,e$ as an unpacking operation, "pattern-matching" on the packed value. Aside from that choice, we have two new values and three new expressions. The values are, respectively, the introduction forms for the constrained arrow and product types: a computation suspended pending a proof of constraint, and a pair consisting of a value and such a proof. However, since we do not introduce proof terms for our notion of provability, we use $\bullet$ in places

$$
\begin{array}{lrcl}
\text{values} & v & ::= & x \mid \langle\rangle \mid \lambda x.\, e \mid \langle v_1, v_2\rangle \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \\
 & & & \mid \Lambda.\, e \mid \mathsf{pack}\, v \mid \mathsf{roll}\, v \mid \lambda \bullet.\, e \mid \langle \bullet, v\rangle \\
\text{expressions} & e & ::= & v \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid v_1\ v_2 \mid \mathsf{proj}_1\, v \mid \mathsf{proj}_2\, v \\
 & & & \mid \mathsf{case}\, v\, [x.\, e_1 \mid y.\, e_2] \mid \mathsf{abort}\, v \mid v * \mid \mathsf{let}\, (*, x) = v\, \mathsf{in}\, e \\
 & & & \mid \mathsf{unroll}\, v \mid \mathsf{abort}\, \bullet \mid v\, \bullet \mid \mathsf{let}\, (\bullet, x) = v\, \mathsf{in}\, e \\
\text{eval. contexts} & E & ::= & \square \mid \mathsf{let}\, x = E\, \mathsf{in}\, e
\end{array}
$$

Figure 2.4: The syntax of expressions and values.

where we might imagine a proof or an assumption. This is in direct correspondence to the treatment of universal and existential types, where witnesses are also elided.

In the same spirit, the "proof application", $v\,\bullet$, and "proof unpacking", $\mathsf{let}\,(\bullet, x) = v\,\mathsf{in}\,e$, serve as elimination forms for the two constructs. This leaves us with the final expression, $\mathsf{abort}\,\bullet$, whose behavior matches the eliminator for an empty type — *i.e.* the program should be considered erroneous whenever the expression is in an evaluation position. Its intended role will become clearer when we consider the type system, presented in Figure 2.5.

The typing judgment assigns a type (of kind T) to an expression or a value in three contexts: Δ, which assigns kinds to type variables, Φ, which lists the constraints we assume to hold, and Γ, which assigns types (of kind T in Δ) to term-level variables. Most rules follow System F_ω, with the additional context passed around; we discuss the other rules below. First, the form of the F_ω rule that allows one to replace the type of a term with any type equivalent to it is slightly different in our calculus, as we use the provability relation for the appropriate constraint, rather than the external type equivalence judgment. This means that the proof may depend on equalities, some of which can be introduced by the derivation — a fact that is crucial to encode pattern matching over GADTs. Second, the rules for values and expressions associated with the constrained types match the intuition of introduction and elimination rule. When type-checking the body of $\lambda\bullet.\, e$ we may assume that any well-formed constraint χ holds, and assign the type $\chi \to \tau$ to the expression — but in order to use a value of this type, through $v\,\bullet$, we need to show that the required constraint does indeed hold. The rules for constrained pairs are analogous. Finally, the rule for $\mathsf{abort}\,\bullet$ requires that a discriminable equality can be proved using our assumptions: in other words, that the assumptions are inconsistent. This rule is crucial, since it allows us to directly encode pattern matching for GADTs in a calculus where all case expressions are, by necessity, exhaustive: in a way, it corresponds to OCaml's "dot pattern", which forces the typechecker to try to ensure that a given case in a pattern-match is impossible due to the equalities that would have to hold. The final extension with respect to the standard System F_ω is the presence of the recursive types; the corresponding rules are marked with a red μ. Due to their higher-kinded nature, the recursive types need to be appropriately applied in order for the roll expression to typecheck; the same arguments are passed to the recursive type's unfolding. As usual, the typing rule for the unroll expression is simply an inverse of the one for roll.

$$\frac{x : \tau \in \Gamma}{\Delta \mid \Phi \mid \Gamma \vdash x : \tau} \qquad \frac{}{\Delta \mid \Phi \mid \Gamma \vdash \langle\rangle : \mathsf{unit}} \qquad \frac{\Delta \vdash \sigma :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma, x : \sigma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \lambda x.\, e : \sigma \to \tau}$$

$$\frac{(\Delta \mid \Phi \mid \Gamma \vdash v_i : \tau_i)_{i \in \{1,2\}}}{\Delta \mid \Phi \mid \Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Delta \vdash \tau_2 :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{inj}_1\, v : \tau_1 + \tau_2}$$

$$\frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau_2}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{inj}_2\, v : \tau_1 + \tau_2} \qquad \frac{\Delta, \alpha :: \kappa \mid \Phi \mid \Gamma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \Lambda.\, e : \forall \alpha :: \kappa.\, \tau}$$

$$\frac{\Delta \vdash \sigma :: \kappa \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau[\sigma/\alpha]}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{pack}\, v : \exists \alpha :: \kappa.\, \tau} \qquad \frac{\kappa = (\kappa_i \Rightarrow)_i\, \mathsf{T} \qquad (\Delta \vdash \sigma_i :: \kappa_i)_i \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau[\mu\alpha :: \kappa.\, \tau/\alpha]\ (\sigma_i)_i}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{roll}\, v : (\mu\alpha :: \kappa.\, \tau)\ (\sigma_i)_i}\ (\mu)$$

$$\frac{\Delta \vdash \chi\ \mathsf{constr} \qquad \Delta \mid \Phi, \chi \mid \Gamma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \lambda \bullet.\, e : \chi \to \tau} \qquad \frac{\Delta \mid \Phi \Vdash \chi \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \langle \bullet, v \rangle : \chi \times \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \tau_1 \equiv_{\mathsf{T}} \tau_2 \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_2}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash e_1 : \sigma \qquad \Delta \mid \Phi \mid \Gamma, x : \sigma \vdash e_2 : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v_1 : \sigma \to \tau \qquad \Delta \mid \Phi \mid \Gamma \vdash v_2 : \sigma}{\Delta \mid \Phi \mid \Gamma \vdash v_1\, v_2 : \tau} \qquad \frac{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_1 \times \tau_2}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{proj}_1\, v : \tau_1}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_1 \times \tau_2}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{proj}_2\, v : \tau_2} \qquad \frac{\Delta \vdash \tau :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \mathsf{void}}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{abort}\, v : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_1 + \tau_2 \qquad (\Delta \mid \Phi \mid \Gamma, x_i : \tau_i \vdash e_i : \tau)_{i \in \{1,2\}}}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{case}\, v\, [x_1.\, e_1 \mid x_2.\, e_2] : \tau} \qquad \frac{\Delta \mid \Phi \mid \Gamma \vdash v : \forall \alpha :: \kappa.\, \tau \qquad \Delta \vdash \sigma :: \kappa}{\Delta \mid \Phi \mid \Gamma \vdash v * : \tau[\sigma/\alpha]}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \exists \alpha :: \kappa.\, \sigma \qquad \Delta \vdash \tau :: \kappa \qquad \Delta, \alpha :: \kappa \mid \Phi \mid \Gamma, x : \sigma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{let}\, (*, x) = v\, \mathsf{in}\, e : \tau}$$

Figure 2.5: Type system of $\mathrm{F}^{=i}_{\omega\mu}$. The roll and unroll rules (marked with a red (μ)) are not considered in $\mathrm{F}^{=i}_{\omega}$. The rules are grouped by separating values and expressions, and, within these two groups, by connectives.

$$\frac{\kappa = (\kappa_i \Rightarrow)_i \,\mathsf{T} \qquad (\Delta \vdash \sigma_i :: \kappa_i)_i \qquad \Delta \mid \Phi \mid \Gamma \vdash v : (\mu\alpha :: \kappa.\ \tau)\ (\sigma_i)_i}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{unroll}\, v : \tau[\mu\alpha :: \kappa.\ \tau/\alpha]\ (\sigma_i)_i}\ (\mu)$$

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \equiv_\kappa \sigma_2 \qquad \Delta \Vdash \sigma_1 \#_\kappa \sigma_2 \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{abort}\bullet : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \chi \to \tau \qquad \Delta \mid \Phi \Vdash \chi}{\Delta \mid \Phi \mid \Gamma \vdash v \bullet : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \chi \times \sigma \qquad \Delta \mid \Phi, \chi \mid \Gamma, x : \sigma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{let}\,(\bullet, x) = v\,\mathsf{in}\,e : \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \tau_1 \equiv_\mathsf{T} \tau_2 \qquad \Delta \mid \Phi \mid \Gamma \vdash e : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash e : \tau_2}$$

Figure 2.5: Type system of $\mathrm{F}^{=i}_{\omega\mu}$. The roll and unroll rules (marked with a red (μ)) are not considered in $\mathrm{F}^{=i}_{\omega}$. The rules are grouped by separating values and expressions, and, within these two groups, by connectives. (cont.)

In order to see these rules in action, consider the type constructor Foo, defined as $\mathsf{Foo} \triangleq \lambda\alpha :: \mathsf{T}.\ (\mathsf{Bool} \times (\alpha \equiv_\mathsf{T} \mathsf{Bool})) + \mathsf{unit}$, and the program: $e \triangleq \mathsf{case}\, x\, [y.\ \mathsf{let}\,(\bullet, y) = y\,\mathsf{in}\,\mathsf{abort}\bullet \,|\, \_.\ \langle\rangle]$. Assuming $x : \mathsf{Foo\ unit}$ — and that Bool and unit are discriminable — we can check that $e : \mathsf{unit}$. This is because in the first branch of the case expression we unpack the constrained pair, introducing a constraint $\mathsf{unit} \equiv_\mathsf{T} \mathsf{Bool}$ as an assumption — but since this is a discriminable equality, the abort instruction typechecks, thus expressing the fact that our program should never reach this point during its evaluation. Admittedly, this example is not particularly useful: we discuss more practical examples and their encodings in the following section.

First, though, we turn to the operational semantics of our programs, presented in Figure 2.6. This, again, is largely standard — and the rules for the new constructs simply force appropriate computations or pass the argument to the computation that requires it. As usual, we model failure by a stuck computation: thus, neither of the abort expressions has any contraction rule associated with it.

Nontermination in $\mathrm{F}^{=i}_{\omega\mu}$ It is clear that well-typed programs in our calculus are not necessarily terminating, due to the inclusion of recursive types. However, it is interesting to observe that recursive types are not the *only* source of non-terminating behavior: the combination of impredicative quantification and injectivity of type-

$$\mathsf{let}\, x = v \,\mathsf{in}\, e \mapsto e[v/x]$$
$$(\lambda x.\ e)\ v \mapsto e[v/x]$$
$$(\mathsf{proj}_i \langle v_1, v_2\rangle \mapsto v_i)_{i\in\{1,2\}}$$
$$(\mathsf{case}\,\mathsf{inj}_i\, v\, [x.\ e_1 \,|\, x.\ e_2] \mapsto e_i[v/x])_{i\in\{1,2\}}$$
$$(\Lambda.\ e)\ * \mapsto e$$
$$\mathsf{let}\,(*, x) = \mathsf{pack}\, v \,\mathsf{in}\, e \mapsto e[v/x]$$
$$\mathsf{unroll}\,(\mathsf{roll}\, v) \mapsto v \ (\mu)$$
$$(\lambda \bullet.\ e)\ \bullet \mapsto e$$
$$\mathsf{let}\,(\bullet, x) = \langle \bullet, v\rangle \,\mathsf{in}\, e \mapsto e[v/x]$$

$$\frac{e_1 \mapsto e_2}{E[e_1] \to E[e_2]}$$

Figure 2.6: Operational semantics: contraction and reduction relations; rules marked with a red (μ) are not considered in $\mathrm{F}^{=i}_{\omega}$

constructors with higher-kinded arguments is sufficient.[4]

To observe this behavior, consider the following type, well-kinded at T if $\alpha :: \mathsf{T}$, for any constructor $c :: (\mathsf{T} \Rightarrow \mathsf{T}) \Rightarrow \mathsf{T}$ (either the existential, universal or recursive type constructor):

$$\tau_c^{\mathrm{loop}} \triangleq \exists \beta :: \mathsf{T} \Rightarrow \mathsf{T}.\ (c\ \beta \equiv_{\mathsf{T}} \alpha) \times (\beta\ \alpha \to \mathsf{void})$$

We can now use the same constructor to close-off our type, taking $\tau_c^{\mathrm{cl}} \triangleq c\ (\lambda \alpha :: \mathsf{T}.\ \tau_c^{\mathrm{loop}})$, and use it to give a type to the following variant of the staple of non-terminating computations:

$$v^{\mathrm{loop}} \triangleq \lambda x.\ \mathsf{let}\,(*, (\bullet, y)) = x \,\mathsf{in}\, y\ (\mathsf{pack}\,\langle \bullet, y\rangle)$$

Indeed, we can easily show that, in closed contexts,

$$\vdash v^{\mathrm{loop}} : \tau_c^{\mathrm{loop}}[\tau_c^{\mathrm{cl}}/\alpha] \to \mathsf{void},$$

taking τ_c^{loop} itself as the existential witness, and reflexivity as the required proof of type equivalence, and using injectivity of c on the assumption retrieved from the package to ensure that y is typed — in both instances — at $\tau_c^{\mathrm{loop}}[\tau_c^{\mathrm{cl}}/\alpha]$. Analogous packaging of v^{loop} will ensure that $\vdash v^{\mathrm{loop}}\ (\mathsf{pack}\,\langle \bullet, v^{\mathrm{loop}}\rangle) : \mathsf{void}$, and we get a closed expression at the empty type whose evaluation does not terminate (since it reduces to itself).

A simpler sub-calculus Since the main object of our study, System $\mathrm{F}^{=i}_{\omega\mu}$, exhibits non-terminating behavior without utilizing the recursive types, the injective, internalized equalities clearly increase the expressive power over standard System F_{ω}.

[4] We adapt a proof of falsehood in Idris via injective type constructors, available at https://github.com/idris-lang/Idris-dev/issues/3687, itself an adaptation of a similar proof in Lean, discussed at https://github.com/leanprover/lean/issues/654. While these systems are significantly different, the main problem — injectivity at higher kinds and impredicativity leading to inconsistency — remains applicable.

However, to make this point even stronger, and to better highlight some of the difficulties in extending relational techniques to systems with GADTs, in Section 2.3 we study a sub-calculus of $F^{=i}_{\omega\mu}$, called $F^{=i}_{\omega}$ and establish its relationship to System F_ω. This removes the recursive types and all the rules associated with them, *i.e.* all the rules marked with (μ) and — in order to avoid the nonterminating construction discussed above — prohibits the use of the injectivity rule on quantifiers (which, in effect, restricts its use to sum, product and arrow types).

2.2.1 Expressing GADTs and Pattern Matching

We now discuss how the features of $F^{=i}_{\omega\mu}$ allow for encoding of common patterns of GADTs. As is folklore, this will be achieved through a combination of existential quantification and equality constraints over types — however, in order to demonstrate how the system operates, we pay special attention to the equality reasoning required, particularly discriminability and injectivity.

As mentioned in the previous section, the distinguishing feature of (proper) GADTs is restriction of the index of certain constructors of a type constructor. Below, we present an alternative encoding of perfectly balanced binary trees, which is uniform in the type of its labels, but uses a separate index to guarantee constant depth on all paths. In the absence of other extensions of the type system, the classic Haskell approach is to define "type-level natural numbers", *i.e.* two distinguishable type constructors, one of kind T, and one of kind $\mathsf{T} \Rightarrow \mathsf{T}$, which we would externally identify with zero and successor. The implementation of these, and the balanced binary tree, are presented in Fig. 2.7.

There are a few things of note in the implementation. First, note that both `Z` and `S a` are empty types — however, since the ML lineage of languages treats data types nominally, they are still distinguishable, and `S` is considered injective. Second, the discriminability of these two types ensures that the two constructors of `GTree` have distinct indices: this, in turn, ensures that both arguments of `NodeG` have the same height, as measured by our pseudo-natural number type that represents the index. Finally, we can write a type-safe `left` function that recovers the left subtree of a *non-empty* tree. The non-emptiness of the argument is enforced by a constraint on its index: the `EmptyG` pattern is impossible, due to discriminability.

Our encoding of this example will not be entirely direct, due to the structural treatment of types in $F^{=i}_{\omega\mu}$. Therefore, we take the type unit as encoding the depth 0, and the type constructor $\lambda\alpha :: \mathsf{T}.\ \mathsf{unit} + \alpha$ as the successor.[5] This ensures that the two "constructors" of type-level natural numbers are discriminable, and that the "successor" is indeed injective. Now we can proceed with the encoding, presented in Fig. 2.8. There are a few things to note here. First, notice that we can take the type of labels, α_t, as an *external* parameter to the recursive type, as it remains uniform throughout the definition. On the other hand, the depth-encoding index, α_n, varies through the

[5] In this example, the precise types do not matter, since their role is *phantom* — they do not influence the computation other than through reasoning about their equality. As we shall see, this is not always the case.

```
data Z where
data S a where

data GTree a n where
  EmptyG :: GTree a Z
  NodeG :: GTree a n -> a -> GTree a n -> GTree a (S n)

left :: GTree a (S n) -> GTree a n
left (NodeG l _ _) = l
```

Figure 2.7: An example GADT, the type of perfectly-balanced trees.

$$\mathsf{GTree} :: \mathsf{T} \Rightarrow \mathsf{T} \Rightarrow \mathsf{T}$$
$$\mathsf{GTree} \triangleq$$
$$\lambda\,\alpha_t :: \mathsf{T}.\ \mu\,\varphi :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\,\alpha_n :: \mathsf{T}.$$
$$((\alpha_n \equiv_{\mathsf{T}} \mathsf{unit}) \times \mathsf{unit})$$
$$+ \exists \alpha'_n ::$$
$$\mathsf{T}.\ (\alpha_n \equiv_{\mathsf{T}} (\alpha'_n + \mathsf{unit})) \times \varphi\ \alpha'_n \times \alpha_t \times \varphi\ \alpha'_n$$

$$\mathsf{left} : \forall \alpha_t, \alpha_n ::$$
$$\mathsf{T}.\ \mathsf{GTree}\ \alpha_t\ (\alpha_n + \mathsf{unit}) \to \mathsf{GTree}\ \alpha_t\ \alpha_n$$
$$\mathsf{left} \triangleq \Lambda.\ \Lambda.\ \lambda x.\ \mathsf{case}\ \mathsf{unroll}\, x$$
$$\mid \mathsf{inj}_1\ (\bullet, \_).\ \mathsf{abort}\, \bullet$$
$$\mid \mathsf{inj}_2\ (*, (\bullet, \langle l, \langle \_, \_ \rangle \rangle)).\ l$$

Figure 2.8: $\mathrm{F}^{=i}_{\omega\mu}$-encoding of `GTree` and `left`. We use syntactic sugar in the definition of left to improve readability.

recursive calls and has to be bound *within* the recursive type. Finally, the body of the type is a disjoint sum, with both branches restricting the index α_n through an equality constraint. The left branch is only available at index unit, while the right branch is available at any index $\alpha'_n + \mathsf{unit}$ — with the new, smaller index α'_n bound existentially, and used at the recursive calls.

In the definition of the left function, we use some syntactic sugar to offset the verbosity of our fine-grain call-by-value notation. We introduce the two type variables, α_t and α_n, and an argument x of type $\mathsf{GTree}\ \alpha_t\ (\alpha_n + \mathsf{unit})$, and pattern-match on its unrolled form. In the left case, we obtain a unit value, and a constraint $\alpha_n + \mathsf{unit} \equiv_{\mathsf{T}} \mathsf{unit}$, which is clearly discriminable: thus, we can use the abort expression to obtain any well-formed type. In the right case, on the other hand, we introduce a fresh type variable α'_n, the existential witness of the depth of our subtrees, and a constraint $\alpha_n + \mathsf{unit} \equiv_{\mathsf{T}} \alpha'_n + \mathsf{unit}$. Note that the variable l, which denotes the left subtree, has type $\mathsf{GTree}\ \alpha_t\ \alpha'_n$, which must not escape the scope of the case: thus, it is crucial to cast the type through an equality to one that is well-formed *outside* the case expression. We can achieve this due to the injectivity rules, since the constraint we introduced implies that $\alpha_n \equiv_{\mathsf{T}} \alpha'_n$ holds. Thus, we have that $\mathsf{left} : \forall \alpha_t :: \mathsf{T}.\ \forall \alpha_n :: \mathsf{T}.\ \mathsf{GTree}\ \alpha_t\ (\alpha_n + \mathsf{unit}) \to \mathsf{GTree}\ \alpha_t\ \alpha_n$, which is the expected type.

In the previous example, the indices at which we used type equalities were entirely separated from the term-level content of the types: they were *phantom*. This afforded us a lot of leeway in terms of the encoding, as we only needed to ensure that appropriate type constructors were injective or discriminable. However, it is not always the case that indices do not carry any semantic meaning — GADTs also can be used to encode

```
data Tm v where
  Lift :: a -> Tm a
  Abs  :: (a -> Tm b)
    -> Tm (a -> b)
  App  :: Tm (a -> b)
    -> Tm a -> Tm b
```

```
eval :: Tm a -> a
eval (Lift x) = x
eval (Abs f) =
  \x -> eval (f x)
eval (App f x) =
  (eval f) (eval x)
```

Figure 2.9: GADT and evaluation of well-formed lambda terms

$$\mathsf{Tm} :: \mathsf{T} \Rightarrow \mathsf{T}$$
$$\mathsf{Tm} \triangleq$$
$$\mu\varphi :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha :: \mathsf{T}.$$
$$\alpha + (\exists\beta,\gamma :: \mathsf{T}.\ (\alpha \equiv_{\mathsf{T}} (\beta \to \gamma)) \times (\beta \to \varphi\ \gamma)) + (\exists\beta :: \mathsf{T}.\ \varphi\ (\beta \to \alpha) \times \varphi\ \beta)$$

$$\mathsf{eval} : \forall\alpha :: \mathsf{T}.\ \mathsf{Tm}\ \alpha \to \alpha$$
$$\mathsf{eval} \triangleq$$
$$\mathsf{fix}\,\lambda f.\ \Lambda.\ \lambda x.\ \mathsf{case}\ \mathsf{unroll}\,x\ |\ \mathsf{inj}_1\,y.\ y\ |\ \mathsf{inj}_2\,y.\ \mathsf{case}\ y\ |\ \mathsf{inj}_1\,(*,(*,(\bullet,g)))).\ \lambda z.\ f * (g\ z)\ |\ \mathsf{inj}_2\,(*,\langle g,x\rangle).\ (f * g)\ (f * x)$$

Figure 2.10: $\mathrm{F}^{=i}_{\omega\mu}$-encoding of `Tm` and `eval`. We use syntactic sugar in the definition of eval to improve readability.

data types whose indices are used in nontrivial manner. In Fig. 2.9 we revisit the well-formed lambda terms example from the introduction: note that the presence of the `Lift` constructor allows us to treat any metalanguage value of type `a` as a constant term of that type. This binds the type structure of the indices to the type structure at the metalevel: we are no longer free to choose the encoding of the other indices arbitrarily, as we would lose the connection to the lifted values at the appropriate index. This can be observed in the well-typed evaluator in Fig. 2.9, which transforms lambda-terms indexed with type `a` into values *of that type*. Although this pattern requires a rather tight connection between the indices of recursive types and the types themselves, we can encode it in $\mathrm{F}^{=i}_{\omega\mu}$ using the same approach presented above. The encodings are presented in Fig. 2.10. Note that there is no equality constraint in the first branch of the sum type: this means the corresponding case in pattern-matching is never discriminable, and thus has to be considered in any program. This matches the intended behavior, as we consider a non-exhaustive pattern matching a type error. Since the definition of eval is recursive, it utilizes a fix combinator, which is definable through the use of recursive types, using the standard construction.

2.3 Non-expressibility of $\mathrm{F}^{=i}_{\omega}$ in F_{ω}

We begin the study of our calculus by establishing its relationship with System F_{ω} — the polymorphic lambda calculus with higher kinds, of which $\mathrm{F}^{=i}_{\omega}$ is an extension. For brevity, we do not present the standard rules of System F_{ω} for kinding, constructor

equivalence, and typing.

It is well-known that System F_ω is strong enough to express some notions of equality using universal quantification at higher kinds. For instance, Atkey [9] uses a Church-style definition of equality as its own eliminator, as the following type constructor, although a Leibniz-style definition is also possible [13, §5.4, Definition 5.4.17]:

$$\mathsf{eq}_\kappa \triangleq \lambda\, \alpha, \beta :: \kappa.\ \forall \rho :: \kappa \Rightarrow \kappa \Rightarrow \mathsf{T}.\ (\forall \gamma :: \kappa.\ \rho\ \gamma\ \gamma) \rightarrow \rho\ \alpha\ \beta$$

This leads to a pertinent question: does the reification of equalities and the complex type system proposed in the previous section add any expressive power, or is it an over-complicated reimagining of F_ω?

To answer this question we study the restricted system, $F_\omega^{=i}$, equipped with reified equality, discriminability and the injectivity rules for type constructors with ground arguments, but without considering the recursive types or injectivity on higher-kinded arguments, which would take the system beyond the expressive power of F_ω by allowing nonterminating behaviors. We study the problem of expressibility of $F_\omega^{=i}$ in F_ω as an extension of Felleisen's [1991] approach to typed languages, *i.e.* the question of existence of a *structural* translation of the internalized equalities and associated constructs, which preserves the structure of the features already present in the target language. Through an appeal to a model of F_ω defined below, we establish that no such translation that preserves typing may exist.

The propositional model of F_ω. The model we construct in this section treats the F_ω types as propositions, in the usual Curry-Howard fashion: the idea is to enforce the condition that the inhabited types are mapped to true propositions, while the empty types are mapped to false ones. Our formalization uses the Coq type of propositions; however, a set-theoretic model of truth is equally valid.

We begin by defining the interpretation of kinds:

$$[\![\mathsf{T}]\!] \triangleq \mathrm{Prop}$$
$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \rightarrow [\![\kappa_2]\!],$$

with the implicit notion that equivalent propositions are equal, and that functions preserve equality and are themselves equal extensionally. Now we can define the interpretation of well-kinded types of System F_ω, which is presented in Fig. 2.11. We take the usual type of the interpretation function as $[\![\Delta \vdash \tau :: \kappa]\!] : (\prod_\alpha [\![\Delta(\alpha)]\!]) \rightarrow [\![\kappa]\!]$, and omit the contexts and kinds to avoid cluttering the definition. Note that the constructors are interpreted as appropriate logical connectives; in particular, the empty type is interpreted as falsehood.

It is immediate that the definition is well-formed. As a soundness result, we obtain the following theorem, where the interpretation of the term-variable context is given as by the conjunction of the interpretations of types.

Theorem 2.3.1. *For any well-typed term $\Delta \mid \Gamma \vdash e : \tau$ in F_ω and any $\eta : \prod_\alpha [\![\Delta(\alpha)]\!]$ we have $[\![\Gamma]\!]_\eta \Rightarrow [\![\tau]\!]_\eta$.*

$$
\begin{array}{rlllll}
[\![\alpha]\!]_\eta & \triangleq \eta(\alpha) & [\![\times]\!]\ \varphi_1\ \varphi_2 & \triangleq \varphi_1 \wedge \varphi_2 & [\![\mathsf{unit}]\!] & \triangleq \top \\
[\![\lambda\alpha :: \kappa.\ \tau]\!]_\eta & \triangleq \lambda\varphi.[\![\tau]\!]_{\eta[\alpha\mapsto\varphi]} & [\![+]\!]\ \varphi_1\ \varphi_2 & \triangleq \varphi_1 \vee \varphi_2 & [\![\mathsf{void}]\!] & \triangleq \bot \\
[\![\sigma\ \tau]\!]_\eta & \triangleq [\![\sigma]\!]_\eta([\![\tau]\!]_\eta) & [\![\to]\!]\ \varphi_1\ \varphi_2 & \triangleq \varphi_1 \Rightarrow \varphi_2 & [\![\forall_\kappa]\!]\ \varphi & \triangleq \forall\psi : [\![\kappa]\!].\ \varphi(\psi) \\
[\![c]\!]_\eta & \triangleq [\![c]\!] & & & [\![\exists_\kappa]\!]\ \varphi & \triangleq \exists\psi : [\![\kappa]\!].\ \varphi(\psi)
\end{array}
$$

Figure 2.11: Propositional interpretation of the types (left) and constructors (middle and right) of F_ω.

Note that since the types in Γ and τ are of kind T, the theorem is well-formed. With this result, we are ready to tackle the problem of expressibility of $\mathrm{F}_\omega^{=i}$.

Non-existence of a translation. Armed with the model of F_ω, we can now prove the following non-expressibility theorem:

Theorem 2.3.2. *There cannot exist a family of translations $\lceil - \rceil : F_\omega^{=i} \to F_\omega$ for expressions, values and types such that the following conditions hold:*

- *$\lceil - \rceil$ preserve closedness of types and terms;*
- *$\lceil - \rceil$ is homomorphic on constructs of F_ω;*
- *if $\cdot \mid \cdot \mid \cdot \vdash e : \tau$ holds in $F_\omega^{=i}$, then $\cdot \mid \cdot \mid \cdot \vdash \lceil e \rceil : \lceil \tau \rceil$ holds in F_ω.*

Note that, while we restrict ourselves to conditions on typing of closed programs with no equality assumptions, the requirement that the translation is homomorphic ensures that the typing also needs to be preserved in open contexts that can arise as application of the typing rules of F_ω. Certainly, the most important feature of a translation from $\mathrm{F}_\omega^{=i}$ to F_ω would be the elimination of constraints and equality assumptions: we make no assumptions as to how this would be achieved, as the impossibility of the translation follows already from its required behavior on types and type constructors.

Proof. Assume that a translation that satisfies the requirements exists, and consider a type constructor $\varphi \triangleq \lambda\alpha :: \mathsf{T}.\ (\alpha \equiv_{\mathsf{T}} \mathsf{unit} + \mathsf{void}) \times \mathsf{unit}$, well-kinded in $\mathrm{F}_\omega^{=i}$ at kind $\mathsf{T} \Rightarrow \mathsf{T}$. Clearly, there exists an expression e, such that $\cdot \mid \cdot \mid \cdot \vdash e : \varphi\ (\mathsf{unit} + \mathsf{unit}) \to \mathsf{void}$, since the assumed constraint allows us (by using injectivity) to cast the unit value to type void. Thus, by the properties of translation, we get that $\cdot \mid \cdot \vdash \lceil e \rceil : \lceil \varphi \rceil\ (\mathsf{unit} + \mathsf{unit}) \to \mathsf{void}$, and consequently, by the properties of the propositional model, $([\![\lceil \varphi \rceil]\!]\ \top) \Rightarrow \bot$.

However, observe that the type $\varphi\ (\mathsf{unit} + \mathsf{void})$ is clearly inhabited (by the value $\langle \bullet, \langle\rangle \rangle$), and thus, by properties of the translation and the soundness of the model, $[\![\lceil \varphi \rceil]\!].\top$ holds, leading to a contradiction. □

Note that the proof depends only on injectivity of the disjoint sum type constructor and the ability to cast through type equalities. An analogous construction can be performed for discriminability, since $\mathsf{unit} + \mathsf{unit}$ and unit are also equal in our model of F_ω. The intuitive content of the proof is that the requirement that the translation behaves homomorphically on type constructors cannot be fulfilled: while in System F_ω, a type constructor that builds a non-empty type from a non-empty argument will do so for *any* non-empty argument, this is not necessarily the case in the presence of discriminable (or injective) equality constraints.

Through this theorem, we have established that $F_\omega^{=i}$ is not directly expressible in F_ω. We believe that this highlights the expressive power of GADTs, which depend heavily on injectivity rules to express the precise pattern-matching constructs. Moreover, it highlights that while the adage that GADTs are existential types and type equalities may be true, one ought to be very precise about what can be proved about the equality of types within the system.

Finally, the propositional model of F_ω serves to highlight the difficulties that any traditional relational interpretations is bound to run into: while the propositional model identifies all inhabited types, the standard relational approach tends to, at least, conflate all the *empty* types. It is also difficult to imagine how one could obtain injectivity for type arguments that appear in a contravariant position in the interpretation, such as for the left-hand side of the arrow type.

2.4 Type Soundness of $F_{\omega\mu}^{=i}$

Having established that our calculus is a non-trivial extension of the known systems, we turn to the question of its type soundness. In this section we apply the classic syntactic methodology of progress and preservation [142]. However, since the type-system of $F_{\omega\mu}^{=i}$ includes a complex system of equality reasoning — in particular, a system that is able to express that assumptions are contradictory — the progress lemma becomes non-trivial. On the other hand, the same fact allows us to state in one concise lemma all the properties that are required to prove canonical forms (also called inversion) lemmas for particular type formers. This lemma states the *consistency* of our proof system.

Lemma 2.4.1 (Consistency)**.** *A discriminable constraint is not provable in an empty context: in other words,* $\emptyset \mid \emptyset \Vdash \tau_1 \equiv_\kappa \tau_2$ *and* $\emptyset \Vdash \tau_1 \#_\kappa \tau_2$ *are contradictory.*

We prove this lemma via a *normalization-by-evaluation* argument [16, 31], which we discuss in more detail in Section 2.5.2; for now, we discuss the methodological implications of the lemma. In the case of $F_{\omega\mu}^{=i}$ its statement follows naturally from having the discriminability judgment as part of the type system used to discharge contradictory equality assumptions — a feature that is rarely, if ever, present: most systems with non-trivial equality of types consider them up to relatively simple syntactic rearrangements or some forms of subtyping. However, we believe that even in those cases it may be a useful methodological approach to define these relations

in such a way that a variant of Lemma 2.4.1 can be stated. Moreover, if the system admits a natural characterization of normal forms and non-trivial computation on the level of types, we believe that normalization-by-evaluation is a natural path to follow in proving consistency. To the best of our knowledge, this is the first time such a consistency property is used explicitly as a crucial ingredient in a Wright–Felleisen-style proof of type soundness via progress and preservation [142].

Lemma 2.4.1 can be used to eliminate impossible cases in progress and preservation lemmas. Its first application is in the proofs of the canonical forms and inversion lemmas, which have to account for possible applications of equality casting rule: here, it plays the role of traditional inversion lemmas in type systems with subtyping principles. The lemma allows us to eliminate the impossible cases, which are trivially discriminable: thus, the notion of discriminability serves to simplify and modularize these proofs. We observe this in the following instance of a canonical form lemma for the arrow types.

Lemma 2.4.2 (Canonical form for arrows)**.** *If v is a closed value of type τ and τ is provably equal to some arrow type in an empty context, then v is a lambda-abstraction with a well-typed body.*

$$(\emptyset \mid \emptyset \Vdash \tau \equiv_{\mathsf{T}} (\tau_1 \to \tau_2)) \wedge (\emptyset \mid \emptyset \mid \Gamma \vdash v : \tau) \implies (\exists x\, e.\, v = \lambda x.\, e \wedge \emptyset \mid \emptyset \mid \Gamma, x : \tau_1 \vdash e : \tau_2)$$

Proof. By induction on the typing derivation. Of the applicable proof rules only type-cast and lambda abstraction rules are not discriminable; the remainder can be dispatched via Lemma 2.4.1 (*e.g.* if $\tau = \tau_3 + \tau_4$). In the first of the remaining cases we proceed by induction, since the type equality is transitive. Finally, for the lambda abstraction we only need to cast the variable and expression through appropriate equalities, which follow from injectivity of the arrow constructor. □

Lemma 2.4.3 (Preservation)**.** *Reductions preserves typing in the sense that for any well-typed $\emptyset \mid \emptyset \mid \emptyset \vdash e : \tau$ such that $e \to e'$, we also have $\emptyset \mid \emptyset \mid \emptyset \vdash e' : \tau$.*

Proof. The proof is by induction on a given typing derivation.[6] All the cases (except for type conversion, which is trivial) follow by case analysis of the reduction, with the appropriate canonical form lemmas used where necessary. □

Lemma 2.4.4 (Progress)**.** *Any well-typed expression $\emptyset \mid \emptyset \mid \emptyset \vdash e : \tau$ can either be reduced or is already a value.*

Proof. The proof proceeds by induction on the typing derivation. All the cases, except for the (trivial) type conversion and discrimination of impossible equalities, follow from canonical form lemmas. The latter case holds due to the contradiction rule for equality constraints. In this case, we assume that $e = \mathsf{abort}\,\bullet$, and both $\emptyset \mid \emptyset \Vdash \tau \equiv_{\mathsf{T}} \sigma$ and $\emptyset \Vdash \tau \#_{\mathsf{T}} \sigma$ hold. By Lemma 2.4.1, we evidently have a contradiction. □

Definition 2.4.5 (Safety)**.** *A closed expression e is called* safe *when any expression that e reduces to is irreducible if and only if it is a value.*

[6] This allows us to omit inversion lemmas for elimination forms.

Theorem 2.4.6 (Soundness)**.** *Any well-typed expression* $\emptyset \mid \emptyset \mid \emptyset \vdash e : \tau$ *is safe in the sense of Definition 2.4.5.*

Proof. The proof follows from progress and preservation lemmas. □

2.5 Relational Models of Injective Type Equalities

In this section we discuss relational models of $\mathrm{F}^{=i}_{\omega\mu}$. We begin by discussing in detail the challenge of constructing models that validate injectivity rules (this challenge could already be observed in Section 2.3). In Section 2.5.2 we follow this by constructing a unary model, which utilizes a normalization-by-evaluation view of the types, and which we can use to semantically justify type soundness. In Section 2.5.3 we discuss the limitations of this construction, in particular for the case of binary relations (as needed for reasoning about representation independence), and, finally, we extend the model construction to allow for semantic binary relations as needed for reasoning about representation independence. We use the model to obtain the first known proofs of representation independence in a setting with generalized algebraic data types.

2.5.1 The Challenge of Injective Relational Interpretations

To appreciate the challenge of constructing a relational interpretation where all type constructors would be injective, consider the propositional interpretation of System F_ω presented in Figure 2.11. While relational interpretations — whether unary or binary — contain additional information, the basic structure remains very similar. In particular, whenever a given type is uninhabited, its interpretation is empty: this leads to the resurgence of the problems that threaten validity of the injectivity and discriminability rules. Therefore, a *direct* interpretation of types as predicates on values is *not* viable.

A universe of codes. A natural attempt to square this circle is through introduction of some universe of *codes* of types, into which we could interpret types in a way that would validate injectivity and discriminability, and which we could decode (or interpret, or realize), in a separate step, as predicates on values. This, however, presents its own challenges, as the structure of the types is rather rich. Indeed, there would be two conflicting requirements: (1) In order to obtain a justification of injectivity and discriminability rules, such a universe (or at least a significant part of it) would need to be inductively defined. But at the same time, (2) the injectivity of the quantifiers and recursive type constructors, and the rich equational theory involving the higher-kinded types, requires a rather semantic view of the codes, particularly at higher types. Fulfilling those seemingly conflicting requirements without succumbing to paradoxes is challenging — and it is the subject of the remainder of this section.

$$\frac{c :: \kappa}{\vdash c :: \mathrm{Neu}^{\Delta}_{\kappa}} \qquad \frac{x :: \kappa \in \Delta}{\vdash x :: \mathrm{Neu}^{\Delta}_{\kappa}} \qquad \frac{\vdash \nu :: \mathrm{Neu}^{\Delta}_{\kappa_a \Rightarrow \kappa_r} \qquad \vdash \mu :: \mathrm{Nf}^{\Delta}_{\kappa_a}}{\vdash \nu\ \mu :: \mathrm{Neu}^{\Delta}_{\kappa_r}}$$

$$\frac{\vdash \chi :: \mathrm{NC}^{\Delta} \qquad \vdash \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}{\vdash \chi \to \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}} \qquad \frac{\vdash \chi :: \mathrm{NC}^{\Delta} \qquad \vdash \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}{\vdash \chi \times \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}$$

$$\frac{\vdash \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}{\vdash \nu :: \mathrm{Nf}^{\Delta}_{\mathsf{T}}} \qquad \frac{\vdash \mu :: \mathrm{Nf}^{\Delta,\alpha::\kappa_a}_{\kappa_r}}{\vdash \lambda\alpha :: \kappa_a.\ \mu :: \mathrm{Nf}^{\Delta}_{\kappa_a \Rightarrow \kappa_r}} \qquad \frac{\left(\vdash \mu_i :: \mathrm{Nf}^{\Delta}_{\kappa}\right)_{i \in \{1,2\}}}{\vdash \mu_1 \equiv_{\kappa} \mu_2 :: \mathrm{NC}^{\Delta}}$$

Figure 2.12: A judgmental presentation of neutral and normal types, and normal constraints.

2.5.2 A Unary Model via Normalization-by-Evaluation

A key insight of the model construction we now present is that the above mentioned requirements for our universe of codes match very closely those known from the area of normalization-by-evaluation: we have the *inductively* defined normal forms, and *semantic* interpretations, which are used to justify the sophisticated equational or computational theory. In our case, the neutral and normal forms of types and constraints are presented in Figure 2.12.

Definition 2.5.1. *We will write $\mathcal{K}$ for the category of kinding contexts Δ with morphisms $\delta : \Delta \to \Delta'$ given by kind-preserving* renamings *sending variables $\alpha \in \Delta'$ to variables $\delta^*\alpha : \Delta'(\alpha)$ in context Δ. We will write $\mathcal{Y}_{\mathcal{K}} : \mathcal{K} \hookrightarrow \mathrm{Pr}(\mathcal{K})$ for the* ***Yoneda embedding*** *of $\mathcal{K}$ into its category of presheaves $\mathrm{Pr}(\mathcal{K}) = \mathbf{Set}^{\mathcal{K}^{\mathrm{op}}}$ sending $\Delta \in \mathcal{K}$ to the representable functor $\hom_{\mathcal{K}}(-, \Delta)$.*

For a given kind κ, the collections of normal constructors, well-formed neutral constructors, and normal constraints form *presheaves* Nf_{κ}, Neu_{κ}, and NC on the category $\mathcal{K}$, as described inductively in Fig. 2.12. For example, $\mathrm{Nf}^{\Delta}_{\kappa}$ is the set of normal forms of constructors of kind κ in context Δ. The functorial action is given by syntactic renaming of variables.

Definition 2.5.2 (Intepretation of kinds)**.** *We now define an NbE-inspired interpretation of kinds into presheaves on the category $\mathcal{K}$ of kinding contexts and renamings as follows:*

$$[\![\mathsf{T}]\!] \triangleq \mathrm{Neu}_{\mathsf{T}}$$
$$[\![\kappa_a \Rightarrow \kappa_r]\!] \triangleq [\![\kappa_a]\!] \Rightarrow [\![\kappa_r]\!]$$

In the second clause above, the right-hand $\Rightarrow$ denotes the exponential presheaf [87, §I.6, Proposition 1]. The interpretation of kinds κ is extended to kind contexts Δ

$$\begin{array}{ll}
\text{reify} : [\![\kappa]\!] \Rightarrow \text{Nf}_\kappa & \text{reflect} : \text{Neu}_\kappa \Rightarrow [\![\kappa]\!] \\
\text{reify}(\nu : [\![\mathsf{T}]\!]) \triangleq \nu & \text{reflect}(\nu : \text{Neu}_\mathsf{T}) \triangleq \nu \\
\text{reify}(\varphi : [\![\kappa_a \Rightarrow \kappa_r]\!]) \triangleq & \text{reflect}(\nu : \text{Neu}_{\kappa_a \Rightarrow \kappa_r})(\mu : [\![\kappa_a]\!]) \triangleq \\
\quad \lambda\alpha :: \kappa_a.\ \text{reify}(\varphi\,(\text{reflect}(\alpha))) & \quad \text{reflect}(\nu\ (\text{reify}(\mu)))
\end{array}$$

Figure 2.13: Reification and reflection functions, defined in the internal language of $\text{Pr}(\mathcal{K})$.

pointwise, using the cartesian product of presheaves:

$$[\![\Delta]\!] \triangleq \prod_{\alpha :: \kappa \in \Delta} [\![\kappa]\!]$$

We use the interpretation of kinds defined above as the type of the interpretation function for types; the interpretation of constraints will target constraints in normal form.

The normalization procedure can now proceed in the usual way, by defining the injection of neutral forms and reification, and reflection of well-formed types into the interpretation of their kinds (and normalization of constraints).

We begin by defining, by mutual induction on the structure of kinds, the *reification* of interpretations of kinds as normal forms and the *reflection* of neutrals as interpretation of kinds (the main purpose of the latter is to perform a semantic analogue of eta-expansion). In order to enforce the well-formedness, these are defined as exponential presheaves; we present the implementation in Figure 2.13, making use of the cartesian closed structure of presheaves via the internal language of $\text{Pr}(\mathcal{K})$. Note how the reification ensures that all functional types will be in eta-long form, and how the variable α needs to be reflected to be passed as an argument to φ, since its kind may be functional (and thus the variable may need to be eta-expanded as well).

Definition 2.5.3 (The identity environment)**.** *For each context* Δ*, we may define an* identity environment $\text{id}_\Delta : [\![\Delta]\!]^\Delta$ *using the reflection operation, setting* $\text{id}_\Delta(\alpha :: \kappa \in \Delta) \triangleq \text{reflect}(\alpha)$.

With reification and reflection in place, we can define the interpretation of types. As with the other two functions, well-formedness with respect to renaming is ensured by taking the denotation in the exponential presheaf:

$$[\![\Delta \vdash \tau :: \kappa]\!] : [\![\Delta]\!] \Rightarrow [\![\kappa]\!]$$

We present the implementation of this interpretation function in Figure 2.14, again using the internal language of $\text{Pr}(\mathcal{K})$.

While this definition bakes in the requirement that the interpretation of our types is well-behaved in terms of functoriality, we need more: due to the presence of injectivity, if the interpretation of our constraints is to verify the reasoning rules, reification needs to be injective, at least on the image of interpretation of types.

$$
\begin{aligned}
[\![\Delta \vdash \alpha :: \kappa]\!]_\eta &\triangleq \eta(\alpha) \\
[\![\Delta \vdash \lambda\alpha :: \kappa_a.\ \tau :: \kappa_a \Rightarrow \kappa_r]\!]_\eta &\triangleq \mu \mapsto [\![\Delta, \alpha :: \kappa_a \vdash \tau :: \kappa_r]\!]_{\eta[\alpha \mapsto \mu]} \\
[\![\Delta \vdash \sigma\ \tau :: \kappa_r]\!]_\eta &\triangleq [\![\Delta \vdash \sigma :: \kappa_a \Rightarrow \kappa_r]\!]_\eta\, [\![\Delta \vdash \tau :: \kappa_a]\!]_\eta \\
[\![\Delta \vdash c :: \kappa]\!]_\eta &\triangleq \mathrm{reflect}(c) \\
[\![\Delta \vdash \chi \to \tau :: \mathsf{T}]\!]_\eta &\triangleq [\![\Delta \vdash \chi\ \mathsf{constr}]\!]_\eta \to [\![\Delta \vdash \tau :: \mathsf{T}]\!]_\eta \\
[\![\Delta \vdash \chi \times \tau :: \mathsf{T}]\!]_\eta &\triangleq [\![\Delta \vdash \chi\ \mathsf{constr}]\!]_\eta \times [\![\Delta \vdash \tau :: \mathsf{T}]\!]_\eta \\
[\![\Delta \vdash \tau_1 \equiv_\kappa \tau_2\ \mathsf{constr}]\!]_\eta &\triangleq \mathrm{reify}([\![\Delta \vdash \tau_1 :: \kappa]\!]_\eta) \equiv_\kappa \mathrm{reify}([\![\Delta \vdash \tau_2 :: \kappa]\!]_\eta)
\end{aligned}
$$

Figure 2.14: Interpretation of types and constraints, specified in the internal language of $\mathrm{Pr}(\mathcal{K})$.

$$
\begin{aligned}
\eta \mid \nu_1 \approx_{\mathsf{T}} \nu_2 &\triangleq [\![\nu_1]\!]_\eta = \nu_2 \\
\eta \mid \varphi_1 \approx_{\kappa_a \Rightarrow \kappa_r} \varphi_2 &\triangleq \forall \Delta_1', \Delta_2', (\delta_1 : \hom_{\mathcal{K}}(\Delta_1', \Delta_1), \delta_2 : \hom_{\mathcal{K}}(\Delta_2', \Delta_2)), (\eta' : [\![\Delta_1']\!]^{\Delta_2'}). \\
&\qquad \forall \mu_1, \mu_2. \big(\delta_2^* \eta = \lambda x.\ \eta'(\delta_1(x))\big) \to \big(\eta' \mid \mu_1 \approx_{\kappa_a} \mu_2\big) \\
&\qquad \to \big(\eta' \mid \varphi_1(\delta_1, \mu_1) \approx_{\kappa_r} \varphi_2(\delta_2, \mu_2)\big)
\end{aligned}
$$

Figure 2.15: A logical relation connecting interpretations of types via an environment. The relation $\eta \mid \mu_1 \approx_\kappa \mu_2$ ranges over $\eta : [\![\Delta_1]\!]^{\Delta_2}$, $\mu_1 : [\![\kappa]\!]^{\Delta_1}$, and $\mu_2 : [\![\kappa]\!]^{\Delta_2}$.

Lemma 2.5.4. *For any types τ_1, τ_2 of kind κ, well-formed in Δ and any* good *(defined in the following paragraph) environment $\eta : [\![\Delta]\!]^{\Delta'}$, if* $\mathrm{reify}([\![\tau_1]\!]_\eta) = \mathrm{reify}([\![\tau_2]\!]_\eta)$*, then* $[\![\tau_1]\!]_\eta = [\![\tau_2]\!]_\eta$.

This lemma does not depend on conventional soundness or completeness lemmas for NbE, nor is it useful in their proofs in the absence of injectivity. However, in the presence of injectivity of constructors, it is crucial to the proof of completeness of NbE, which needs to justify the reasoning rules of Fig. 2.3 — in particular, the injectivity rule. There, we can assume that $[\![c\ (\sigma_i)_i]\!]_\eta = [\![c\ (\tau_i)_i]\!]_\eta$, and need to prove that $([\![\sigma_i]\!]_\eta = [\![\tau_i]\!]_\eta)_i$. The types in our assumption are applications of a constructor to arguments, and thus neutral, but this only allows us to establish that $(\mathrm{reify}([\![\sigma_i]\!]_\eta) = \mathrm{reify}([\![\tau_i]\!]_\eta))_i$: thus, to establish completeness, we need Lemma 2.5.4.

To prove this lemma, we introduce a novel logical relation $(\eta \mid \mu_1 \approx_\kappa \mu_2)$, as depicted in Fig. 2.15. Note that, in contrast to the usual relation that appears in the proof of soundness and relates syntactic view of types to the semantic view, this relation connects two semantic types via a mediating environment, which serves to reconcile the free variables in the two types.[7] The relation naturally extends to a pair of environments (with matching domain) related through a third, mediating

[7] To the best of our knowledge, this approach has not been previously employed in the NbE literature.

environment. We say that a semantic type $\mu \in [\![\kappa]\!]^\Delta$ (respectively, environment) is *good* when it is related to itself via the identity environment:

$$\mathrm{good}(\mu) \triangleq (\mathrm{id}_\Delta \mid \mu \approx_\kappa \mu)$$

With this definition, we can establish the (limited) injectivity of reification that we need via a series of technical lemmas. The most important include the following pair of results.

Lemma 2.5.5. *If* $\eta \mid \mu_1 \approx \mu_2$*, then* $[\![\mathrm{reify}(\mu_1)]\!]_\eta = \mu_2$.

Lemma 2.5.6. *If* $\eta \mid \eta_1 \approx \eta_2$*, then* $\eta \mid [\![\tau]\!]_{\eta_1} \approx [\![\tau]\!]_{\eta_2}$.

The first of these establishes that we can obtain the related semantic type via reinterpretation of the reification; the second ensures that interpreting a type with related environments yields related results.

With Lemma 2.5.4, we can finally establish correctness of equational reasoning and discriminability.

Lemma 2.5.7. *If* $\Delta \mid \Phi \Vdash \psi$ *holds, and* $\eta : [\![\Delta]\!]^{\Delta'}$ *is a good environment and* $[\![\varphi]\!]_\eta$ *is true for any* $\varphi \in \Phi$*, then* $[\![\psi]\!]_\eta$ *is also true. Moreover, if* $\Delta \Vdash \tau_1 \#_\kappa \tau_2$ *holds, then* $[\![\tau_1 \equiv_\kappa \tau_2]\!]_\eta$ *is false.*

In the above, we say that (a normal form of) a constraint is true if the two types (in normal forms) are identical. As a corollary, we obtain the consistency lemma of Section 2.4.

Realizing NbE interpretations as predicates on values. While the normalization-by-evaluation argument has given us *some* interpretation of types that can be used to justify the equality and discriminability rules of the calculus, it is far from obvious, *a priori*, that we can use it to give a relational interpretation on values. To get an intuitive view of what remains to be done, note that we have largely dealt with the problem of open types, and the main problem that remains is that of open terms. Note also that, crucially, it is enough to consider evaluation in closed contexts, both for terms and types, and that we only need to give a relational interpretation to types of kind T. Therefore, we define a function $\mathcal{R} : \mathrm{Neu}^{\cdot}_{\mathsf{T}} \to \mathrm{Val} \to \mathit{iProp}$, using a mixture of guarded and structural recursion. Here *iProp* is a universe of step-indexed propositions, which comes equipped with a later modality and supports definition of recursive predicates when the recursion is guarded by the later modality (concretely, *iProp* can be understood as downwards-closed sets of natural numbers or as the universe of propositions in an appropriate logic framework, such as LSLR [36] or Iris [73]). The definition is presented in Figure 2.16. Note that, in contrast to the usual presentation of impredicative quantification, we *need* to use guarded recursion for universal and existential quantifiers to ensure that the interpretation is well-defined, as the normal form of the body of the quantified type needs to be *reinterpreted* to obtain a normal form of the resulting type. Thus, this approach does not conflict

$$
\begin{aligned}
\mathcal{E}(P)(e) &\triangleq (\exists v.\, P(v) \wedge e = v) \vee \left(\exists e'.\, e \to e' \wedge \triangleright \mathcal{E}(P)(e')\right) \\
\mathcal{R}(\mathsf{unit})(v) &\triangleq v = \langle\rangle \\
\mathcal{R}(\mathsf{void})(v) &\triangleq \bot \\
\mathcal{R}(\nu_1 \times \nu_2)(v) &\triangleq \exists v_1, v_2.\, v = \langle v_1, v_2\rangle \wedge \bigwedge_{i \in \{1,2\}} \mathcal{R}(\nu_i)(v_i) \\
\mathcal{R}(\nu_1 + \nu_2)(v) &\triangleq \bigvee_{i \in \{1,2\}} \exists v'.\, v = \mathsf{inj}_i\, v' \wedge \mathcal{R}(\nu_i)(v') \\
\mathcal{R}(\nu_1 \to \nu_2)(v) &\triangleq \forall u.\, \mathcal{R}(\nu_1)(u) \to \mathcal{E}(\mathcal{R}(\nu_2))(v\, u) \\
\mathcal{R}(\forall \alpha :: \kappa.\, \tau)(v) &\triangleq \exists e.\, v = \Lambda.\, e \wedge \forall \mu \in [\![\kappa]\!](\cdot).\, \mathrm{good}(\mu) \\
&\qquad \to \triangleright \mathcal{E}(\mathcal{R}([\![\tau]\!]_{[\alpha \mapsto \mu]}))(e) \\
\mathcal{R}(\exists \alpha :: \kappa.\, \tau)(v) &\triangleq \exists v'.\, v = \mathsf{pack}\, v' \wedge \exists \mu \in [\![\kappa]\!](\cdot).\, \mathrm{good}(\mu) \\
&\qquad \wedge \triangleright \mathcal{R}([\![\tau]\!]_{[\alpha \mapsto \mu]})(v') \\
\mathcal{R}((\mu \alpha :: \kappa.\, \tau)\, \overline{\sigma})(v) &\triangleq \exists v'.\, v = \mathsf{roll}\, v' \wedge \triangleright \mathcal{R}([\![(\tau[\mu \alpha :: \kappa.\, \tau/\alpha])\, \overline{\sigma}]\!].)(v') \\
\mathcal{R}(\chi \to \nu)(v) &\triangleq \chi\, \mathsf{true} \to \mathcal{E}(\mathcal{R}(\nu))(v\, \bullet) \\
\mathcal{R}(\chi \times \nu)(v) &\triangleq \exists v'.v = \langle \bullet, v'\rangle \wedge \chi\, \mathsf{true} \wedge \mathcal{R}(\nu)(v')
\end{aligned}
$$

Figure 2.16: The interpretation of closed, neutral ground types as predicates and an evaluation closure; both notions are defined via guarded recursion.

with the non-terminating example of Section 2.2: the fact that the model validates the injectivity rules for universal and existential quantifiers precludes an interpretation of the quantifiers that doesn't require us to count a computation step.

With the interpretation of the closed types defined (in two stages), we can now turn to the definition of the logical relation, which is presented in Figure 2.17. We pay attention to the fact that the environments for the type-variable context Δ are "good", *i.e.* that all the interpretations are self-related, which ensures that this property is inherited by any interpretation of a type with that environment. The environment needs to validate all the equality assumptions in Φ, and the closing environment is the standard extension of the relation on values. With this definition, it is easy to prove the fundamental theorem of logical relations:

Theorem 2.5.8 (Fundamental)**.** *Any well-typed value,* $\Delta \mid \Phi \mid \Gamma \vdash v : \tau$*, is in the logical interpretation of its type,* i.e. $\Delta \mid \Phi \mid \Gamma \models v : \tau$*. Any well-typed expression,* $\Delta \mid \Phi \mid \Gamma \vdash e : \tau$*, is in the logical interpretation of its type for computations,* i.e. $\Delta \mid \Phi \mid \Gamma \models e : \tau$.

Proof. By induction on the structure of the derivation, using appropriate compatibility lemmas. For the type conversion rules, note that the assumptions of Lemma 2.5.7 are

$$
\begin{aligned}
[\![\Phi]\!]_\eta \,\mathrm{true} &\triangleq \forall \varphi \in \Phi.\ [\![\varphi]\!]_\eta \,\mathrm{true} \\
[\![\Gamma]\!]_\eta &\triangleq \{\gamma \in \mathrm{dom}(\Gamma) \to \mathrm{Val} \mid \forall x \in \mathrm{dom}(\Gamma).\ \mathcal{R}([\![\Gamma(x)]\!]_\eta)(\gamma(x))\} \\
\Delta \mid \Phi \mid \Gamma \models v : \tau &\triangleq \forall \eta \in [\![\Delta]\!](\cdot).\ \mathrm{good}(\eta) \to [\![\Phi]\!]_\eta \,\mathrm{true} \to \forall \gamma \in [\![\Gamma]\!]_\eta \\
&\qquad \to \mathcal{R}([\![\tau]\!]_\eta)(v[\gamma]) \\
\Delta \mid \Phi \mid \Gamma \models e : \tau &\triangleq \forall \eta \in [\![\Delta]\!](\cdot).\ \mathrm{good}(\eta) \to [\![\Phi]\!]_\eta \,\mathrm{true} \to \forall \gamma \in [\![\Gamma]\!]_\eta \\
&\qquad \to \mathcal{E}(\mathcal{R}([\![\tau]\!]_\eta))(e[\gamma])
\end{aligned}
$$

Figure 2.17: The definition of the logical relation.

satisfied. Thus, the normal forms of the two types are equal, and so their realizers coincide. □

Limitations of the model based on the NbE interpretation of types. While the model presented in this section justifies semantically the type system of System $\mathrm{F}^{=i}_{\omega\mu}$, and the realizability interpretation can be naturally extended to the binary case, with closed, neutral ground types interpreted as step-indexed relations rather than predicates, the interpretation has an important shortcoming. The limiting factor stems from the fact that all our semantic types can be reified as (a subset of) *well formed, syntactic types*. Thus, the universally or existentially quantified variables can only be instantiated with such types. (Concretely, for universal types, *e.g.*, the relation τ in Figure 2.16 is in $[\![\kappa]\!](\cdot)$.) In the binary case, this clearly prevents us from reasoning about representation independence for abstract data types, as we are obliged to *choose* the representation once and for all — and even in the unary case, it prevents us from proving safety of a (syntactically) ill-typed implementation of an interface by semantic means. In effect, by *allowing* the sophisticated equational reasoning with types whose syntactic structure (or part of it) must be known, we have severely circumscribed the model. So the question is: Can we relax these constraints for types whose syntactic structure we need not know and then obtain a more powerful semantic model supporting reasoning about semantic typing (in the unary case) and representation independence (in the binary case)? The answer is yes, as we now explain.

2.5.3 A Relaxed Model for Representation Independence

We now show that the NbE model *can* be relaxed — albeit by relinquishing the canonical property that the inductively defined "normal forms" are a subset of well-formed types. Thus, rather than defining the neutral and normal forms of types, we define *open universes* (still split into neutral and normal) at each kind. These universes, presented in Figure 2.18, are largely as before, since we still need to enforce injectivity of our type constructors, except that the neutral universe at ground kind also contains the entire space of uniform relations on closed values! These are considered distinct

$$\frac{\varphi : \mathrm{Val}^2 \to \mathit{iProp}}{\varphi : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}} \qquad \frac{c :: \kappa}{c : \mathcal{U}_{\kappa}^{\natural\Delta}} \qquad \frac{x :: \kappa \in \Delta}{x : \mathcal{U}_{\kappa}^{\natural\Delta}} \qquad \frac{\nu : \mathcal{U}_{\kappa_a \Rightarrow \kappa_r}^{\natural\Delta} \qquad \mu : \mathcal{U}_{\kappa_a}^{\Delta}}{\nu\ \mu : \mathcal{U}_{\kappa_r}^{\natural\Delta}}$$

$$\frac{\chi : \mathcal{U}_{\mathsf{C}}^{\Delta} \qquad \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}{\chi \to \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}} \qquad \frac{\chi : \mathcal{U}_{\mathsf{C}}^{\Delta} \qquad \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}{\chi \times \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}$$

$$\frac{\nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}{\nu : \mathcal{U}_{\mathsf{T}}^{\Delta}} \qquad \frac{\mu : \mathcal{U}_{\kappa_r}^{\Delta, \alpha :: \kappa_a}}{\lambda\, \alpha :: \kappa_a.\ \mu : \mathcal{U}_{\kappa_a \Rightarrow \kappa_r}^{\Delta}} \qquad \frac{\left(\mu_i : \mathcal{U}_{\kappa}^{\Delta}\right)_{i \in \{1,2\}}}{\vdash \mu_1 \equiv_{\kappa} \mu_2 : \mathcal{U}_{\mathsf{C}}^{\Delta}}$$

Figure 2.18: Open universes, generalizing the neutral and normal forms of types and normal constraints. As with normal forms, we define three universes by mutual induction: the *neutral* universe $\mathcal{U}_{\kappa}^{\natural\Delta}$, the *normal* universe $\mathcal{U}_{\kappa}^{\Delta}$ and the universe of *normal constraints* $\mathcal{U}_{\mathsf{C}}^{\Delta}$; as before, the functorial action is given by syntactic renaming.

from all the other, more syntactic, members of the universe and are considered equal when equivalent as (uniform) relations. Note that our universes remain functorial with respect to the type variable contexts, as the relations are inert, unchanging under the context morphism action.

We can now replay the same construction as presented in the previous section, with one key difference: where the reinterpretation of semantic types used to piggyback on the (implicit) coercion of neutral and normal forms to well-formed types, we cannot do so here. This, however, is not a problem: we replay the definition of Figure 2.14 twice: once for types (targeting our open universes), and once for elements of the universe themselves. An analogue of the logical relation in Figure 2.15 then lets us establish that Lemma 2.5.7 holds in this interpretation as well.

With the generalized NbE-style model, we can now give its realization (for closed, ground, neutral universe) as relations on values. The definition is presented in Figure 2.19; the construction proceeds in an entirely analogous fashion, with the main difference being the interpretation of the relation φ — which is simply realized as itself. The other minor difference is that the components of the recursive type have to be interpreted on their own and applied as the elements of the semantic interpretation, rather than coerced into a single type; this does not lead to any significant differences.

The remainder of the interpretation is standard: we interpret open types by quantifying over an environment of closing environments. The difference with respect to the simpler model is that the interpretations for open type variables (as well as the ones in the realizability interpretation of universal and existential quantifiers) now allow a closed relation on values to be picked. This allows picking safe-but-syntactically-ill-typed implementations (in the unary case), as well as implementations that relate different implementations (in the binary case), thus allowing us to recover some data abstraction capacities. Note that for this to be viable, the context cannot depend on the structure of the picked interpretation (through constraints): this is the cost of

$$
\begin{aligned}
\mathcal{E}(P)(e_1,e_2) &\triangleq (\exists v_1,v_2.\ e_1 = v_1 \wedge e_2 \to^* v_2 \wedge P(v_1,v_2)) \vee \\
&\quad \left(\exists e_1',e_2'.\ e_1 \to e_1' \wedge e_2 \to^* e_2' \wedge \triangleright \mathcal{E}(P)(e_1',e_2')\right) \\
\mathcal{R}(\varphi)(u,v) &\triangleq \varphi(u,v) \\
\mathcal{R}(\mathsf{unit})(u,v) &\triangleq u = v = \langle\rangle \\
\mathcal{R}(\mathsf{void})(u,v) &\triangleq \bot \\
\mathcal{R}(\nu_1 \times \nu_2)(u,v) &\triangleq \exists u_1,u_2,v_1,v_2.\ u = \langle u_1,u_2\rangle \wedge v = \langle v_1,v_2\rangle \wedge \bigwedge_{i\in\{1,2\}} \mathcal{R}(\nu_i)(u_i,v_i) \\
\mathcal{R}(\nu_1 + \nu_2)(u,v) &\triangleq \bigvee_{i\in\{1,2\}} \exists u',v'.\ u = \mathsf{inj}_i\, u' \wedge v = \mathsf{inj}_i\, v' \wedge \mathcal{R}(\nu_i)(u',v') \\
\mathcal{R}(\nu_1 \to \nu_2)(u,v) &\triangleq \forall u',v'.\ \mathcal{R}(\nu_1)(u',v') \to \mathcal{E}(\mathcal{R}(\nu_2))(u\,u', v\,v') \\
\mathcal{R}(\forall\alpha :: \kappa.\ \tau)(u,v) &\triangleq \exists e,e'.\ u = \Lambda.\ e \wedge v = \Lambda.\ e' \wedge \\
&\qquad \forall \mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \to \triangleright \mathcal{E}(\mathcal{R}([\![\tau]\!]_{[\alpha\mapsto\mu]}))(e,e') \\
\mathcal{R}(\exists\alpha :: \kappa.\ \tau)(u,v) &\triangleq \exists u',v'.\ u = \mathsf{pack}\, u' \wedge v = \mathsf{pack}\, v' \wedge \\
&\qquad \exists \mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \wedge \triangleright \mathcal{R}([\![\tau]\!]_{[\alpha\mapsto\mu]})(u',v') \\
\mathcal{R}(\mu_\kappa\ \varphi\ \overline{\sigma})(u,v) &\triangleq \exists u',v'.\ u = \mathsf{roll}\, u' \wedge v = \mathsf{roll}\, v' \wedge \triangleright \mathcal{R}([\![\varphi]\!].\ [\![\mu_\kappa\ \varphi]\!].\ \overline{[\![\sigma]\!].})(u',v') \\
\mathcal{R}(\chi \to \nu)(u,v) &\triangleq \chi\,\mathsf{true} \to \mathcal{E}(\mathcal{R}(\nu))(u\,\bullet, v\,\bullet) \\
\mathcal{R}(\chi \times \nu)(u,v) &\triangleq \exists u',v'.u = \langle \bullet, u'\rangle \wedge v = \langle \bullet, v'\rangle \wedge \chi\,\mathsf{true} \wedge \mathcal{R}(\nu)(u',v')
\end{aligned}
$$

Figure 2.19: The realization of $\mathcal{U}^{\natural\cdot}_{\mathsf{T}}$ as relations on values.

working in an environment that can reason about the types and depend on non-trivial information about their structure.

Representation independence for non-empty lists. To illustrate that our model supports reasoning about representation independence, we provide a small synthetic example, in which we show contextual equivalence of two implementations of an existentially typed module for non-empty lists of natural numbers, with operations for obtaining the head of a list, constructing a list with a single element, and for inserting an element into a list:

$$\mathsf{nelist} \triangleq \exists\alpha :: \mathsf{T}.\ (\alpha \to \mathbb{N}) \times (\mathbb{N} \to \alpha) \times (\mathbb{N} \to \alpha \to \alpha)$$

The two implementations we consider are lists of natural numbers and vectors of natural numbers. The vectors are implemented as a GADT; they are indexed by their size, which we represent as type-level natural numbers. The implementation details can be found in Figure 2.20. It is worth noting that the GADT enables the definition of a total head operation for vectors.

We use our logical relation to show that two implementations $I_1 = \mathsf{pack}\,\langle \mathsf{head}, \langle \mathsf{inj}, \mathsf{cons}\rangle\rangle$ and $I_2 = \mathsf{pack}\,\langle \mathsf{vhead}, \langle \mathsf{vinj}, \mathsf{vcons}\rangle\rangle$ are contextually

$\mathsf{natlist} :: \mathsf{T}$
$\mathsf{natlist} \triangleq \mu\alpha :: \mathsf{T}.\ \mathsf{unit} + (\mathbb{N} \times \alpha)$

$\mathsf{cons} : \mathbb{N} \to \mathsf{natlist} \to \mathsf{natlist}$
$\mathsf{cons}\, x\, xs \triangleq \mathsf{roll}\, \mathsf{inj}_2\, \langle x, xs \rangle$

$\mathsf{inj} : \mathbb{N} \to \mathsf{natlist}$
$\mathsf{inj}\, x \triangleq \mathsf{cons}\, x\, (\mathsf{roll}\, \mathsf{inj}_1\, \langle\rangle)$

$\mathsf{head} : \mathsf{natlist} \to \mathbb{N}$
$\mathsf{head}\, xs \triangleq$
  $\mathsf{case}\ \mathsf{unroll}\, xs$
  $\mid \mathsf{inj}_1\, \_.\ \mathsf{diverge}$
  $\mid \mathsf{inj}_2\, \langle y, \_ \rangle.\ y$

$\mathsf{natvec} :: \mathsf{T} \Rightarrow \mathsf{T}$
$\mathsf{natvec} \triangleq$
  $\mu\varphi :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha :: \mathsf{T}.$
  $((\alpha \equiv_{\mathsf{T}} \mathsf{void}) \times \mathsf{unit})$
  $+ (\mathbb{N} \times \exists\beta :: \mathsf{T}.\ (\alpha \equiv_{\mathsf{T}} (\beta + \mathsf{unit})) \times (\varphi\, \beta))$

$\mathsf{nenatvec} :: \mathsf{T}$
$\mathsf{nenatvec} \triangleq \exists\alpha :: \mathsf{T}.\ \mathsf{natvec}\, (\alpha + \mathsf{unit})$

$\mathsf{vcons} : \mathbb{N} \to \mathsf{nenatvec} \to \mathsf{nenatvec}$
$\mathsf{vcons}\, x\, xs \triangleq$
  $\mathsf{let}\, (*, ys) = xs\, \mathsf{in}$
  $\mathsf{pack}\, \mathsf{roll}\, \mathsf{inj}_2\, \langle x, \mathsf{pack}\, \langle \bullet, ys \rangle \rangle$

$\mathsf{vinj} : \mathbb{N} \to \mathsf{nenatvec}$
$\mathsf{vinj}\, x \triangleq \mathsf{vcons}\, x\, \mathsf{roll}\, \mathsf{inj}_1\, \langle \bullet, \langle\rangle \rangle$

$\mathsf{vhead} : \mathsf{nenatvec} \to \mathbb{N}$
$\mathsf{vhead}\, xs \triangleq$
  $\mathsf{let}\, (*, ys) = xs\, \mathsf{in}$
  $\mathsf{case}\ \mathsf{unroll}\, ys$
  $\mid \mathsf{inj}_1\, (\bullet, w).\ \mathsf{abort}\, \bullet$
  $\mid \mathsf{inj}_2\, \langle y, \_ \rangle.\ y$

Figure 2.20: $\mathrm{F}^{=i}_{\omega\mu}$-encoding of lists and non-empty type-indexed vectors. We use syntactic sugar throughout.

equivalent[8] at the type nelist. The key step in the proof is to choose an appropriate relation for the existentially quantified type variable α in the nelist type. We use the following relation, which specifies that both lists must be non-empty, and their respective head elements must be related as natural numbers:

$$\{(v_1, v_2) \mid v_1 = \mathsf{roll}\, \mathsf{inj}_2\, \langle u_1, u_2 \rangle \wedge v_2 = \mathsf{pack}\, \mathsf{roll}\, \mathsf{inj}_2\, \langle u_3, u_4 \rangle \wedge \mathcal{R}[\![\mathbb{N}]\!](u_1, u_3)\} \subseteq \mathrm{Val}^2.$$

Parametricity Our model is also strong enough to encompass a number of the classic consequences of parametricity. To demonstrate that, we present a simple free theorem: we show that any value v of type $\forall\alpha :: \mathsf{T}.\ \alpha \to \alpha$ approximates the identity function $\Lambda.\ \lambda x.\ x$.

Proof. It suffices to show that for any $\eta \in [\![\Delta]\!](\cdot)$ such that $\mathrm{good}(\eta)$ and $[\![\Phi]\!]_\eta$ hold, and any substitutions $(\gamma, \gamma') \in [\![\Gamma]\!]_\eta$,

$$\mathcal{R}([\![\forall\alpha :: \mathsf{T}.\ \alpha \to \alpha]\!])_\eta(v[\gamma], (\Lambda.\ \lambda x.\ x)[\gamma']).$$

Given the assumption that $\Delta \mid \Phi \mid \Gamma \vdash v : \forall\alpha :: \mathsf{T}.\ \alpha \to \alpha$, by the fundamental lemma we get $\mathcal{R}([\![\forall\alpha :: \mathsf{T}.\ \alpha \to \alpha]\!])_\eta(v[\gamma], v[\gamma'])$. By the definition of the value interpretation for universal types, $v[\gamma] = \Lambda.\ e$ and $v[\gamma'] = \Lambda.\ e'$.

[8] For brevity, we have not included the standard argument showing that logical relatedness implies contextual approximation; it is included in the Coq formalization. To show contextual equivalence, we show logical relatedness in both directions.

Moreover, $\forall\mu.\triangleright\mathcal{E}(\mathcal{R}([\![\alpha\to\alpha]\!]_{\eta,\alpha\mapsto\mu}))(e,e')$. We instantiate this interpretation with an empty relation. From that we can conclude that either e diverges, in which case the statement trivially holds, or e terminates at a value f. If e terminates at f, then, it suffices to prove the following statement:

$$\forall u\ v\ \mu.\triangleright(\mathcal{R}([\![\alpha]\!]_{\eta,\alpha\mapsto\mu})(u,v)\to\mathcal{E}(\mathcal{R}([\![\alpha]\!]_{\eta,\alpha\mapsto\mu}))(f\ u,v)).$$

We instantiate the interpretation from the fundamental lemma with a singleton relation $\{(u,v)\}\subseteq \mathrm{Val}\times\mathrm{Val}$. From that and determinism of operational semantics, we can conclude that either $t\triangleq f\ u$ terminates or diverges. If it terminates, the result is equal to u by our choice of the relation, in which case the statement trivially holds. If t diverges, then the statement vacuously holds. □

Note that the proof follows the standard pattern, thus providing evidence that the known results can be transferred to a calculus with GADTs. While the example we show here does not explicitly use GADTs, the approach scales: the usual theorems, such as map-map fusion or the free theorems of Wadler [138], can be proved not just for lists, but also types that do use equality constraints, such as vectors or perfect binary trees indexed with type-encoded depth. Note, however, that these properties reflect the payload type of the data structure (which behaves parametrically, even though the structure itself uses equality constraints), rather than the index encoding length or depth, which is much more constrained.

Power and limitations of the relational model As the examples above demonstrate, our model is powerful enough to transport parametricity and representation independence results from simpler systems, without internalized, injective equalities. It is natural to ask whether *all* such results can be preserved in our model, and whether GADTs introduce *new* opportunities for exploiting parametricity.

For the first question, we are aware of certain limitations of our current model. These stem from the fact that we only allow inert relations as semantic elements of our open universes. Thus, we have no way of treating higher-kinded parameters semantically. Thus, for instance, we cannot use an ill-typed but well-behaved implementation of lists as a type constructor of kind $\mathsf{T}\Rightarrow\mathsf{T}$ (over the type of elements). In contrast, concrete instantiation of such a list constructed at known element type do fit into the setup as presented. Whether this limitation can be lifted is one of the questions left for future work.

As for the interaction of GADTs with parametricity, it is clear that type constructors with equality-constrained indices cannot be parametric in those indices: this much is enforced by the model. However, this does not mean that GADTs as a feature are orthogonal to representation independence, free theorems, etc.: the fact that they *can* be used as implementations of abstract data types, used as instances of free theorems, etc., with correct behavior enforced by virtue of the type system is a consequence of the model we have presented in this section.

2.6 Formalization

With the exception of the final part of the non-expressibility theorem in Section 2.3, all the constructions presented in this paper have been formalized in the Coq proof assistant. In this section we discuss some aspects of the formalization that we believe are of wider interest, either from a technical or a methodological standpoint.

2.6.1 Internal Languages *vs.* Explicit Presheaves

In the informal presentation of our results, we have made liberal use of the *internal language* of presheaf categories where possible. For instance, many important constructions involved in our NbE–style model lie in the category $\mathrm{Pr}(\mathcal{K})$ of presheaves on the category $\mathcal{K}$ of kind contexts and kind-preserving renamings. The most direct and intuitive presentation of these constructions, then, uses the internal language of $\mathcal{K}$ as a (locally) cartesian closed category. The use of the internal language is well-established as a way to avoid the proliferation of administrative parameters (quantifying over future worlds, renamings, *etc*.) that routinely obscure what are in essence simple constructions based on simple ideas.

The downside of our use of the internal language is that different parts of our development must take place in different categories (languages), corresponding to the boundary between the $\mathrm{Pr}(\mathcal{K})$ and **Set**. Whereas informal mathematical practice is highly optimized for working rigorously with such abstraction boundaries, the same does not currently hold of proof assistants like Coq which do not yet support mixing different type theories in the same development. For this reason, our Coq formalization explicitly unfolds the internal constructions to external ones; this process introduces many additional verification conditions (*e.g*. naturality laws, *etc*.) that are automatic in the internal presentation, but which we had to check explicitly in our Coq development.

2.6.2 Representation of Terms and Binders

The problem of representing abstract syntax with binding in theorem provers spawned a rich literature, whose review is beyond the scope of this paper. Here, we briefly note that the approach we use is inspired by Fiore et al.'s [1999] algebraic view of abstract syntax and variable binding. In particular, both terms and types are constructed as *presheaves*, *i.e*. `Type`-valued functors on a category of renamings. This is achieved by parameterizing the Coq type of terms with a *type* of its variables, and — through the use of an appropriate library of typeclasses — equipping the resulting type constructor with a functorial structure. We then equip the functor with a monadic structure that encodes substitution.

2.6.3 Normalization by Evaluation: Methodological and Technical Features

The NbE implementation is the foundation of most of the technical developments of the paper: both the syntactic, progress-and-preservation proof of type soundness and the model construction rely on a variant of the procedure. It is, however, also interesting from the technical, proof-engineering point of view — we leverage the representation of syntax based on presheaves over names to interpret types as presheaves. This has the advantage of baking a lot of the requisite structure of types and their normal forms into the construction, and thus removing a large part of the reasoning based on partial equivalence relation that are necessary in less precise encodings in order to prune the semantic spaces of ill-behaved functions. Compare for instance the work of Allais et al. [4], whose interpretation of types, while similar at object level, does not require functoriality: while their implementation may be simpler, it requires significantly more post-hoc reasoning, some of which is unnecessary if we interpret the types into spaces with richer structure. Thus, while the particular NbE implementation we consider is relatively simple, and the categorical notions that we utilize are fairly well-known in this context [46], we believe that our formalization goes further towards an NbE that is correct by construction than the state of the art.

2.6.4 Relational Models and Step-Indexed Logics

The logic used in Section 2.5 is not the standard Coq logic; in particular, it internalizes the step-indexed methods through the use of the later operator and the Löb induction to define and reason about predicates. Thus we had to choose a framework that allows reasoning in such a logic.

Since the requirements of the model were only related to constructing relations via guarded recursion, we decided to use Polesiuk's IxFree library [102], which provides features roughly equivalent to the LSLR logic [36]. Alternative choices are available, however: one is to work in an axiomatic extension of Coq's type theory, like Sterling and Harper [124] — although this does come at a price of the final soundness results being expressed in a different type theory. The other obvious candidate is the Iris framework [73]: its power to construct solutions of recursive domain equations (via its invariants feature) would be useful in extending the calculus to include higher-order mutable state. We chose against using it, for this project, due to its higher complexity, and the fact that in a simpler logic, such as the one provided by IxFree, we can make a much more direct use of the host logic (*i.e.* Coq's) proof manipulation primitives.

2.7 Related Work

Of the many calculi that were developed to account for GADTs and similar features, the one most closely related to our work is System F_C [127, 140]; the calculus developed as an intermediate language for the Glasgow Haskell Compiler. While it accounts for features that we do not handle, such as type functions and equality

axioms, which are part of GHC, the treatment of GADTs is much closer to the surface language, through explicitly defined type constructors and their constructors on term level. The presence of equality constraints leads to similar problems in the syntactic type soundness, and necessitates considering the problem of consistency of the equational theory. However, while the calculus is perfectly well suited as an intermediate language, it is less ideal for the semantic study of the GADTs.

Relational models of higher-kinded polymorphic lambda calculi have been widely studied since the 1990s [57, 105]. Two examples of interest include the formalized developments of Atkey [9] and Vytiniotis and Weirich [137]. Atkey formalizes pure System F_ω, but studies type equality expressible within the system, and inductive types that arise as initial algebras of positive functors. However, the notion of functoriality he uses is external to the calculus, and the encodings of GADTs he obtains through the encoding of Johann and Ghani [66] does not seem to allow discrimination based on distinct types. Instead, he extends F_ω with additional kinds, including type-level natural numbers — which can express some GADT patterns.

Vytiniotis and Weirich build a relational model of an extension of System F_ω with a single built-in GADT [137], and show parametricity properties of this language. We explore a larger class of programs that can contain arbitrary GADTs, including the runtime-type representation type R. The nontermination of their System R_ω when the representation type is extended to include universal quantifiers is likely related to our example in Section 2.2, although the details of the construction differ somewhat. An interesting feature of their model is the inclusion of syntactic information in the interpretation of types and type equivalence, which is quite different from our two-stage interpretation. However, since the closure under type equivalence is only enforced post-hoc, it is not likely that their construction could be scaled to all type constructors being injective, rather than just the built-in representation GADT.

The most recent line of work on semantics and parametricity of GADTs comes from Johann et al. [65, 67, 68, 70]. These are categorical models for languages where type constructors are considered functorial, and inductive types can be formed out of strictly positive functors, as appropriate initial algebras. Our calculus is somewhat more modest, in that the type constructors have no additional structure; on the other hand, we consider an impredicative system with general recursive types, and thus our approach to obtaining relational models must be somewhat different.

A complementary line of work on the syntax and elaboration of GADTs is that of Dunfield and Krishnaswami [38], who have given a proof-theoretic account of equality constraints in terms of the Girard–Schroeder-Heister elimination rule for equality [54, 106], which says that $\Gamma, c \equiv_\kappa d \vdash \mathcal{J}$ holds if and only if $\theta^*\Gamma \vdash \theta^*\mathcal{J}$ holds for all substitutions θ in the complete set of unifiers for $c, d :: \kappa$. Unifiability of two constructors is a syntactic and meta-level concept that does not correspond to any behavior of semantic models; indeed, most languages have constructor injectivity as an admissible rule, but it is a special feature of GADTs for these injectivity laws to be *derivable* and thus required in models. Dunfield and Krishnaswami [38] achieve these laws all at once by incorporating syntactic unifiability into their formal theory; we achieve something analogous in our setting by means of explicit rules. This choice is

important because our main results pertain to semantic models, a viewpoint that is not readily accessed from the Girard–Schroeder-Heister perspective.

2.8 Conclusion

In this work, we developed a core calculus that can be used for semantic study of generalized algebraic data types. We show that the calculus is expressive enough to encode many of the commonly used programming patterns, and that it is strictly more expressive than calculi that do not enforce injectivity and discriminability of constant type constructors. We prove that the calculus is type-safe, and build a novel, two-stage relational model that uses a variation on the normalization-by-evaluation construction to enforce the injectivity and discriminability rules. The model is strong enough to allow for data-abstraction reasoning, the first such for a calculus equipped with GADTs.

Several directions of future work are apparent. First, integrating other advanced language features, such as higher-order mutable state or effect handlers, within our calculus and its model would be an interesting direction that could lead to models that account for virtually all main features of modern functional programming languages; the difficulties in modeling these seem largely orthogonal to those we faced in the current work, so such an extension should be manageable. Second, it seems that even with the inclusion of "inert" step-indexed predicates in the universe, the model is not robust enough to ensure termination of the sub-calculus with restricted injectivity and no recursive types, System $\mathrm{F}_{\omega}^{=i}$ — a property that, we believe, ought to hold. Therefore, we would like to further refine our model, so that it would allow for a more fine-tuned interpretation of non-injective quantifiers, as well as more refined data-abstraction principles. Finally, a possible direction would consider adapting the functorial approach of Johann *et al.* to a setting with a wider space of type constructors. This could allow us to enrich practical programming languages with type constructors that have the mapping action inherently associated with them, including the ability to quantify over such constructs.

CHAPTER 3

Context-Dependent Effects in Guarded Interaction Trees

Abstract

Guarded Interaction Trees are a structure and a fully formalized framework for representing higher-order computations with higher-order effects in Coq. We present an extension of Guarded Interaction Trees to support formal reasoning about context-dependent effects. That is, effects whose behaviors depend on the evaluation context, e.g., `call/cc`, `shift` and `reset`. Using and reasoning about such effects is challenging since certain compositionality principles no longer hold in the presence of such effects. For example, the so-called "bind rule" in modern program logics (which allows one to reason modularly about a term inside a context) is no longer valid. The goal of our extension is to support representation and reasoning about context-dependent effects in the most painless way possible. To that end, our extension is conservative: the reasoning principles (and the Coq implementation) for context-independent effects remain the same. We show that our implementation of context-dependent effects is viable and powerful. We use it to give direct-style denotational semantics for higher-order programming languages with `call/cc` and with delimited continuations. We extend the program logic for Guarded Interaction Trees to account for context-dependent effects, and we use the program logic to prove that the denotational semantics is adequate with respect to the operational semantics. This is achieved by constructing logical relations between syntax and semantics inside the program logic. Additionally, we retain the ability to combine multiple effects in a modular way, which we demonstrate by showing type soundness for safe interoperability of a programming language with delimited continuations and a programming language with higher-order store.

3.1 Introduction

Despite a lot of recent progress, representing and reasoning about programming languages in proof assistants, such as Coq, is still considered a major challenge. The

design space is wide and many approaches have been considered. Recently, research on a novel point in the design space was initiated with the introduction of Interaction Trees [145], or ITrees for short. ITrees were introduced to simplify representation and reasoning about possibly non-terminating programs with side effects in Coq. In a sense, ITrees provide a target for denotational semantics of programming languages, which allows one to abstract from syntactic details often found in models based on operational semantics. ITrees specifically allow one to easily represent and reason about various effects and their combinations in a modular way. A wide range of subsequent applications of ITrees (see, e.g., [78, 115, 146, 147], among others) show that they indeed work very well for representing and reasoning about first-order programs with first-order effects. As part of the trade-offs, ITrees could not support higher-order representations and higher-order effects. To address this challenge, Guarded Interaction Trees (or GITrees for short) were introduced [49].

While GITrees support *higher-order effects*, in particular, the challenging case of higher-order store, they are limited to effects that do not alter the control flow of the program (more specifically, the continuation). As a consequence, one cannot use GITrees to give direct-style denotational semantics of programming languages with context-dependent effects such as `call/cc`, exceptions, or delimited continuations. It is of course possible to give semantics to, e.g., `call/cc` using a CPS translation, but this would require a global transformation which complicates representation and reasoning, especially in combination with other effects present. In this paper we extend GITrees to support direct-style representation and reasoning about higher-order programs with higher-order context-dependent effects in Coq, and evaluate its modularity.

We want to stress that our extension to context-dependent effects is not only theoretically interesting, but also important for scalability, since many real mainstream programming languages include context-dependent effects. Indeed, exceptions are now a standard feature in many languages, and while other context-dependent effects such as delimited continuations are not as widespread in mainstream programming languages, they are present in the core calculi used for some such languages: for example, the Glasgow Haskell Compiler core language was recently extended with delimited continuations to support the introduction of effect systems, which can be efficiently developed on top of delimited continuations [75]. Moreover, effect handlers, which rely on control flow operators, have recently been introduced in OCaml 5.0, and as a design feature in Helium [19], Koka [84, 85], and other languages. Note that these languages do not only include forms of delimited continuations, but also other effects, which underscores the importance of considering delimited control in combination with other effects.

Overview of technical development and key challenges. Similarly to how ITrees are defined as a coinductive type in Coq, GITrees are defined as a guarded recursive type (this is to support function spaces in GITrees; in the presence of function spaces there are negative occurrences of the recursive type and hence one cannot simply

define the type of GITrees using coinduction). Coq does not directly support guarded recursive types, so GITrees are defined using a fragment of guarded type theory implemented in Coq, as part of the Iris framework [62]. To work efficiently with GITrees we make use of other Iris features like separation logic and Iris Proof Mode [79], which we use to define custom program logics for different (combinations of) effects. This enables us to reason about GITrees smoothly in the Iris logic in Coq, in much the same way as one works directly in Coq. We recall the precise definition of GITrees in Section 3.2.

Broadly speaking, GITrees model effects in the following way. The type of GITrees is parameterized over a set of effectful operations. Each operation is given meaning by a *reifier* function, using a form of state monad. From this, we define the *reduction* relation of GITrees, which gives semantics to computations represented by GITrees.

To support context-dependent effects, we extend (Section 3.3) the notion of a reifier so that reification of effects can also depend on the context; technically, the reifier operation becomes parameterized by a suitable GITrees continuation. This extension allows us to give semantics to context-dependent effects, but it comes at a price. In particular, following the change in semantics, we need to reformulate the program logic for GITrees: in the presence of context-dependent effects (like `call/cc`), the so-called "bind"-rule becomes unsound. Of course, we do not want this reformulation to complicate reasoning about computations that do *not* include context-dependent effects. To that end, we parameterize the GITrees (and the program logic) by a flag, which allows us to recover the original proof rules and make sure that all of the original GITrees framework still works with our extension.

To motivate the extension to context-dependent effects, we give direct-style denotational semantics to a higher-order programming language $\lambda_{\mathsf{callcc}}$ with `call/cc` (Section 3.3). Furthermore, we use the derived program logic to construct a logical relation between the denotational and the operational semantics to prove computational adequacy of our model.

Our main interest, however, lies in the treatment of delimited continuations. In Section 3.4 we show how to represent delimited continuations as effects in GITrees, and we use them to define a novel denotational semantics for a programming language with `shift` and `reset` operators. We prove that our denotational semantics is sound with respect to the operational semantics (given by an extension of the CEK abstract machine). We additionally use the program logic to define a logical relation, and prove computational adequacy and *semantic* type soundness. We recall that semantic type soundness is interesting because it allows one to combine syntactically well-formed programs with syntactically ill-typed, but semantically well-behaved programs [133].

As we mentioned, it is important to consider delimited continuations not only on their own, but in combinations with other effects. And indeed, one of the key points of ITrees, and therefore also of GITrees, is that they support reasoning about effecting and language interoperability by establishing a common unifying semantic framework. In this paper, we consider (in Section 3.5) an example of such interaction: we show a type-safe embedding of λ_{delim} with delimited continuations into a language λ_{embed}

with higher-order store. We allow λ_{delim} expressions to be embedded into λ_{embed} by surrounding them by simple glue code, and use a type system to ensure type safety of the combined language. To define the semantics of the combined language we rely on the modularity of GITrees, and combine reifers for delimited continuations with reifiers for higher-order store. We prove type safety of the combined language by constructing a logical relation and use the program logic both to define the logical relation and to verify the glue code between the two languages. The type system for the combined language naturally requires that the embedded code is well-typed according to the type system for λ_{delim} and thus we can rely on the type soundness of λ_{delim} (proved in the earlier Section 3.4) when proving type safety for the combined language. At the end of Section 3.5, we give an example of how to verify a more involved interaction of effects, albeit without the type system.

Summary of Contributions. In summary, we present:

- A conservative (with respect to the old results) extension to GITrees for representing and reasoning about context-dependent effects (Section 3.3).
- A sound and adequate model of a calculus with `call/cc` and `throw`, implemented in a direct style (Section 3.3.3).
- A sound and adequate model of a calculus with delimited continuations, with operations `shift` and `reset`, implemented in a direct style (Section 3.4).
- A type system for interoperability between a programming language with delimited continuations and a programming language with higher-order store, with a semantic type safety proof (Section 3.5).

All results in the paper have been formalized in Coq as a modification to the GITrees library and the previously proved results have been ported to our extension. We conclude and discuss related work in Section 3.6. Before we go on with the main part of the paper, we recall some background material on GITrees.

3.2 Guarded Interaction Trees

In this section we provide an introduction to guarded interaction trees. Our treatment is brief, and we refer the reader to the original paper for details [49].

Iris and Guarded Type Theory. Guarded Interaction trees (GITrees) are defined in Iris logic. Here we briefly touch Iris, and refer the reader to the literature on Iris [73] and guarded type theory [24] for more in-depth details. Iris is a separation logic framework built on top of a model of *guarded type theory*, the main use of which is to solve recursive equations and define guarded recursive types, such as the type of GITrees described below. Moreover, Iris has a specialized proof mode [79], implemented in Coq. This allows the users of Iris to carry out formal reasoning in

$$\tau ::= \mathsf{iProp} \mid 0 \mid \mathbf{1} \mid \mathbb{B} \mid \mathsf{Nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \blacktriangleright \tau \mid I \mid \Sigma_{\mathtt{i}\in\mathtt{I}}\tau_{\mathtt{i}} \mid \Pi_{\mathtt{i}\in\mathtt{I}}\tau_{\mathtt{i}} \mid \ldots$$

$$t ::= x \mid F(t_1,\ldots,t_n) \mid \mathsf{abort}\, t \mid () \mid (t,t) \mid \pi_i\, t \mid \lambda x : \tau.t \mid$$
$$\mathsf{inj}_i\, t \mid \mathsf{match}\, t\, \mathsf{with}\, \overline{\mathsf{inj}_i\, x.t}\, \mathsf{end} \mid \mathsf{next}(t) \mid \mathit{fix}_\tau \mid \ldots$$

$$P ::= \mathsf{False} \mid t =_\tau t \mid P \vee P \mid P \to P \mid \forall x : \tau.P \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \boxed{P} \mid \triangleright P \mid \ldots$$

Figure 3.1: Grammar for the Iris base logic.

$$\begin{aligned}
&\mathsf{guarded\ type}\ \mathbf{IT}_E(A) = \mathsf{Ret} : A \to \mathbf{IT}_E(A)\\
&\quad \mid \mathsf{Fun} : \blacktriangleright(\mathbf{IT}_E(A) \to \mathbf{IT}_E(A)) \to \mathbf{IT}_E(A)\\
&\quad \mid \mathsf{Err} : \mathtt{Error} \to \mathbf{IT}_E(A)\\
&\quad \mid \mathsf{Tau} : \blacktriangleright\mathbf{IT}_E(A) \to \mathbf{IT}_E(A)\\
&\quad \mid \mathsf{Vis} : \prod_{\mathtt{i}\in\mathtt{I}} \left(\mathit{Ins}_{\mathtt{i}}(\mathbf{IT}_E(A)) \times (\mathit{Outs}_{\mathtt{i}}(\mathbf{IT}_E(A)) \to \blacktriangleright\mathbf{IT}_E(A))\right) \to \mathbf{IT}_E(A)
\end{aligned}$$

Figure 3.2: Guarded datatype of interaction trees.

separation logic as if they are proving things normally Coq, as we have done in the formalization of this work. For this reason, in the paper we will work with Iris and its type theory informally. Still, we need to say a few things about the foundations.

The syntax of Iris, shown in Figure 3.1, contains types, terms, and propositions. The grammar is standard for higher-order logic, with the exception of the guarded types fragment, and separation logic connectives. The type of propositions is denoted iProp. The *guarded* part of guarded type theory is the "later" modality $\blacktriangleright$. Intuitively, we view all types as indexed by a natural number, where τ_n contains elements of τ "at time" n. Then $\blacktriangleright\tau$ contains elements of τ at a later time; that is, $(\blacktriangleright\tau)_n = \tau_{n-1}$. There is an embedding $\mathsf{next} : \tau \to \blacktriangleright\tau$, and there is a *guarded* fixed point combinator $\mathit{fix}_\tau : (\blacktriangleright\tau \to \tau) \to \tau$, similar to the unguarded version in PCF. We can also lift functions to $\blacktriangleright$: given $f : A \to B$, we have $\blacktriangleright f : \blacktriangleright A \to \blacktriangleright B$.

For the proposition, Iris contains the usual separation logic connectives, and the two modalities: "later" $\triangleright$ and "persistently" $\Box$. The propositional $\triangleright$ modality reflects the type-level later modality $\blacktriangleright$ on the level of propositions, as justified by the following rule: $\triangleright(\alpha =_\tau \beta) \dashv\vdash \mathsf{next}(\alpha) =_{\blacktriangleright\tau} \mathsf{next}(\beta)$. The persistence modality $\Box P$ states that the proposition P is available without claiming any resources (as it normally is the case in separation logic); crucially it makes the proposition duplicable: $\Box P \vdash (\Box P) * (\Box P)$. An example of a persistent proposition is the invariant proposition $\boxed{P}$, which satisfies $\boxed{P} \vdash \Box\boxed{P}$.

Guarded Interaction Trees. Guarded recursive datatypes are datatypes obtained from recursive equations of the form $X = F(\blacktriangleright X)$. In other words, guarded recursive datatypes are similar to the regular datatypes you see in normal programming languages, but every recursive occurrence of the type must be guarded by the $\blacktriangleright$ modality. The datatype we are concerned with here is the type of GITrees, shown in Figure 3.2. It is parameterized over two types: the ground type A and the effect signature E (more on it below).

Guarded Interaction Trees represent computational trees in which the leaves are of the ground type ($\mathsf{Ret}(a)$), error states ($\mathsf{Err}(e)$), and functions ($\mathsf{Fun}(f)$). The leaves $\mathsf{Ret}(a)$ and $\mathsf{Fun}(f)$ are also called *values*, and we write $\mathbf{IT}^v_E(A)$ for the type of $\mathbf{IT}_E(A)$-values.

The nodes of the computation trees are of the two kinds. The first one is a "silent step" constructor $\mathsf{Tau}(\alpha)$. It represents an unobservable internal step of the computation. For convenience, we use the function $\mathsf{Tick} \triangleq \mathsf{Tau} \circ \mathsf{next} : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ that "delays" its argument. This function satisfies the following equation: $\mathsf{Tick}(\alpha) = \mathsf{Tick}(\beta) \dashv\vdash \triangleright(\alpha = \beta)$.

The second kind of nodes are effects given by $\mathsf{Vis}_i(x,k)$. The parameters I, *Ins* and *Outs* are part of the effect signature E. The set I is the set of *names* of operations. The *arities* of an operation $i \in I$ are given by functors Ins_i and $Outs_i$. Let us give an example.

Consider the following signature for store effects. The signature E_{state} consists of effects $\{\mathtt{write}, \mathtt{read}, \mathtt{alloc}\}$ with the following input/output arities:

$$Ins_{\mathtt{write}}(X) \triangleq Loc \times \blacktriangleright X \qquad Ins_{\mathtt{read}}(X) \triangleq Loc \qquad Ins_{\mathtt{alloc}}(X) \triangleq \blacktriangleright X$$
$$Outs_{\mathtt{write}}(X) \triangleq \mathbf{1} \qquad Outs_{\mathtt{read}}(X) \triangleq \blacktriangleright X \qquad Outs_{\mathtt{alloc}}(X) \triangleq Loc$$

For example, `write` expect a location and a new GITree as its input, and simply returns the unit value as an output. We write $\mathsf{Vis}_{\mathtt{write}}((\ell, \alpha), \lambda\_.\, \beta)$ for the computation that invokes the `write` effect with arguments ℓ and α, waits for it to return, and proceeds as β. Thus, the first argument for Vis_i is the input, and the second one is the continuation dependent on the output. This continuation determines the branching in (G)Itrees.

For effects like above, it is usually convenient to provide wrappers:

$$\mathsf{Alloc}(\alpha : \mathbf{IT}, k : Loc \to \mathbf{IT}) \triangleq \mathsf{Vis}_{\mathtt{alloc}}(\mathsf{next}(\alpha), \mathsf{next} \circ k)$$
$$\mathsf{Read}(\ell : Loc) \triangleq \mathsf{Vis}_{\mathtt{read}}(\ell, \lambda x.x)$$
$$\mathsf{Write}(\ell : Loc, \alpha : \mathbf{IT}) \triangleq \mathsf{Vis}_{\mathtt{write}}((\ell, \mathsf{next}(\alpha)), \lambda x.\mathsf{next}(\mathsf{Ret}(\mathsf{inj}())))$$

When the signature and the return type are clear from the context, we simply write $\mathbf{IT}$ and $\mathbf{IT}^v$ for the GITrees and GITree-values.

Equational theory. GITrees come with a number of operations (defined using the recursion principle) that are used for writing and composing computations. Here we list some of those operations which we will be using. The function $\mathsf{get\_val}(\alpha, f : \mathbf{IT}^v \to \mathbf{IT})$ are used for sequencing computations, and its corresponding equations

$$\begin{aligned}
\mathsf{get_val}(\mathsf{Ret}(a), f) &= f(\mathsf{Ret}(a)) & \mathsf{get_val}(\mathsf{Tau}(t), f) &= \mathsf{Tau}(\blacktriangleright \mathsf{get_val}(t, f)) \\
\mathsf{get_val}(\mathsf{Fun}(g), f) &= f(\mathsf{Fun}(g)) & \mathsf{get_val}(\mathsf{Tick}(\alpha), f) &= \mathsf{Tick}(\mathsf{get_val}(\alpha, f)) \\
\mathsf{get_val}(\mathsf{Err}(e), f) &= \mathsf{Err}(e) & \mathsf{get_val}(\mathsf{Vis}_i(x,k), f) &= \mathsf{Vis}_i(x, \blacktriangleright \mathsf{get_val}(-, f) \circ k)
\end{aligned}$$

Figure 3.3: Example function on Guarded Interaction Trees.

are shown in Figure 3.3. Intuitively, $\mathsf{get\_val}(\alpha, f)$ first tries to compute α to a value (a $\mathsf{Ret}(a)$ or a $\mathsf{Fun}(g)$), and then calls f on that value. Similarly, $\mathsf{get\_fun}(\alpha : \mathbf{IT}, f : \blacktriangleright(\mathbf{IT} \to \mathbf{IT}) \to \mathbf{IT})$ and $\mathsf{get\_ret}(\alpha : \mathbf{IT}_E(A), f : A \to \mathbf{IT}_E(A))$ first compute α to a value; if that value is a function $\mathsf{Fun}(g)$ (resp., $\mathsf{Ret}(a)$), then it proceeds with $f(g)$ (resp., $f(a)$). Otherwise it results in an runtime error.

Crucially, to work with higher-order computations, GITrees provide the "call-by-value" application $\alpha \bullet \beta$ satisfying the following equations:

$$\begin{aligned}
\alpha \bullet \mathsf{Tick}(\beta) &= \mathsf{Tick}(\alpha \bullet \beta) & \alpha \bullet \mathsf{Vis}_i(x,k) &= \mathsf{Vis}_i(x, \lambda y.\, \mathsf{next}(\alpha)\ (\blacktriangleright\bullet)\ k\ y) \\
\mathsf{Tick}(\alpha) \bullet \beta_v &= \mathsf{Tick}(\alpha \bullet \beta_v) & \mathsf{Vis}_i(x,k) \bullet \beta_v &= \mathsf{Vis}_i(x, \lambda y.\, k\ y\ (\blacktriangleright\bullet)\ \mathsf{next}(\beta_v)) \\
\mathsf{Fun}(\mathsf{next}(g)) \bullet \beta_v &= \mathsf{Tick}(g(\beta_v)) & \alpha \bullet \beta &= \mathsf{Err}(\mathit{RunTime}) \text{ in other cases}
\end{aligned}$$

where $- \ (\blacktriangleright\bullet)\ -$ is defined as the lifting of $- \bullet -$ to $\blacktriangleright\mathbf{IT}_E(A) \to \blacktriangleright\mathbf{IT}_E(A) \to \blacktriangleright\mathbf{IT}_E(A)$, and $\beta_v \in \mathbf{IT}^v_E(A)$ is either $\mathsf{Ret}(a)$ or $\mathsf{Fun}(g)$.

The application function $\alpha \bullet \beta$ simulates strict function application. It first tries to evaluate β to a value β_v. Then it tires to evaluate α to a function f. If it succeeds, then it invokes $f(\beta_v)$. If at any point it fails, application results in a runtime error.

For the often-used case of GITrees where the ground type includes natural numbers, we use the function $\mathsf{NatOp} : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to IT \to IT \to IT$ which lifts binary functions on natural numbers to binary functions on GITrees. That is, $\mathsf{NatOp}_f(\alpha, \beta)$ first evaluate GITrees β and α to values. If those values are natural numbers, then it computes f of those numbers and returns the result as a GITree. Otherwise, it returns a runtime error $\mathsf{Err}(\mathit{RunTime})$.

Reification and reduction relation. The semantics for effects are given in terms of *reifiers*. A reifier for the signature E is a tuple (State, r), where *State* is a type representing the internal state needed to reify the effects, and r is a reifier function of the type given in Figure 3.4. The idea is that r_i uses the internal state *State* to compute the output of the effect i based on its input.

For example, for the store effects we take *State* to be a map from locations to $\blacktriangleright\mathbf{IT}$ (representing the heap); and we define the following reifier functions:

$$\begin{aligned}
r_{\mathtt{write}}((\ell, \alpha), \sigma) &= ((), \sigma[\ell \mapsto \alpha]) && \text{(where } \ell \in \sigma\text{, and None otherwise)} \\
r_{\mathtt{read}}(\ell, \sigma) &= (\alpha, \sigma) && \text{(where } \sigma(\ell) = \alpha\text{, and None otherwise)} \\
r_{\mathtt{alloc}}(\alpha, \sigma) &= (\ell, \sigma[\ell \mapsto \alpha]) && \text{(where } \ell \notin \sigma\text{)}
\end{aligned}$$

$$r : \prod_{\mathtt{i} \in E} \mathit{Ins}_{\mathtt{i}}(\mathbf{IT}_E) \times \mathit{State} \to \mathit{option}(\mathit{Outs}_{\mathtt{i}}(\mathbf{IT}_E) \times \mathit{State})$$

$$\frac{r_i(x,\sigma) = (y,\sigma') \qquad k\ y = \mathsf{next}(\beta)}{\mathsf{reify}(\mathsf{Vis}_i(x,k),\sigma) = (\mathsf{Tick}(\beta),\sigma')} \qquad \frac{r_i(x,\sigma) = \mathsf{None}}{\mathsf{reify}(\mathsf{Vis}_i(x,k),\sigma) = (\mathsf{Err}(\mathit{RunTime}),\sigma)}$$

Figure 3.4: Signature of reifiers and the reification function

Given reifiers for all the effects, we define a function $\mathsf{reify} : \mathbf{IT} \times \mathit{State} \to \mathbf{IT} \times \mathit{State}$ (as in Figure 3.4) that, given (α,σ) reifies the top-level effect in α using the state σ, and returns the reified GITree and the updated state.

The reify function is then used to give reduction semantics for GITrees. We write $(\alpha,\sigma) \rightsquigarrow (\beta,\sigma')$ for such a reduction step. The definition of $\rightsquigarrow$ is given internally in the logic:

$$\begin{aligned}(\alpha,\sigma) \rightsquigarrow (\beta,\sigma') \triangleq\ & \big(\alpha = \mathsf{Tick}(\beta) \wedge \sigma = \sigma'\big) \\ & \vee \big(\exists i\, x\, k.\, \alpha = \mathsf{Vis}_i(x,k) \wedge \mathsf{reify}(\alpha,\sigma) = (\mathsf{Tick}(\beta),\sigma')\big)\end{aligned}$$

That is, either α is a "delayed" computation $\mathsf{Tick}(\beta)$, which then reduces to β; or it is an effect that can be reified. Recall that we write Tick for the composition $\mathsf{Tau} \circ \mathsf{next}$.

Note that the reify function operates on the top-level effect of the GITree. But what if the top-level constructor is not Vis, e.g. if we have an effect inside an "evaluation context"? The role of evaluation contexts in GITrees is played by *homomorphisms*, which also allow us to bubble up necessary effects to the top of the GITree.

Definition 3.2.1 (Homomorphism)**.** *A map $f : \mathbf{IT} \to \mathbf{IT}$ is a homomorphism, written $f \in \mathit{Hom}$, if it satisfies:*

$$f(\mathsf{Err}(e)) = \mathsf{Err}(e) \quad f(\mathsf{Tick}(\alpha)) = \mathsf{Tick}(f(\alpha)) \quad f(\mathsf{Vis}_i(x,k)) = \mathsf{Vis}_i(x, \blacktriangleright f \circ k)$$

For example, $\lambda x.\, \alpha \bullet x$ is a homomorphism, and so is $\lambda x.\, \mathsf{get\_val}(x,f)$. On the other hand, $\lambda x.\, \mathsf{Vis}_{\mathtt{alloc}}(\mathsf{next}(x),k)$ (for some fixed k) is *not* a homomorphism.

Program logic. In order to reason about GITrees, we employ the full power of the Iris separation logic framework. The program logic operates on the propositions of the form $\mathsf{wp}\,\alpha\,\{\Phi\}$. This weakest precondition proposition intuitively states that the GITree α is safe to reduce, and when it fully reduces, the resulting value satisfies the predicate Φ. Another important predicate is $\mathsf{has\_state}(\sigma)$, which signifies ownership of the current state σ.

In Figure 3.5 we show the rules, on which we focus in this work. Let us describe their meaning. The rule WP-REIFY allows us to symbolically execute effects in GITrees. It is given in a general form, and is used to derive domain-specific rules for concrete effects. Another important rule is WP-HOM which allows one to separate the reasoning about the computation from the reasoning about the context. The reason why WP-HOM is sound (this is going to be important in the next section when we make it unsound),

$$\textsc{wp-reify}\quad \frac{\mathsf{has\_state}(\sigma) \quad \mathsf{reify}(\mathsf{Vis}_i(x,k),\sigma) = (\mathsf{Tick}(\beta),\sigma') \quad \triangleright\left(\mathsf{has\_state}(\sigma') \mathrel{-\!\!*} \mathsf{wp}\,\beta\,\{\Phi\}\right)}{\mathsf{wp}\,\mathsf{Vis}_i(x,k)\,\{\Phi\}} \qquad \textsc{wp-hom}\quad \frac{f \in \mathit{Hom} \quad \mathsf{wp}\,\alpha\,\{\beta_v.\,\mathsf{wp}\,f(\beta_v)\,\{\Phi\}\}}{\mathsf{wp}\,f(\alpha)\,\{\Phi\}}$$

Figure 3.5: Selected weakest precondition rules.

is because the reduction $\rightsquigarrow$ of GITrees satisfies the following properties which allow one disentangle a homomorphism from the GITree it's applied to:

Lemma 3.2.2. *Let f be a homomorphism. Then,*

- $(\alpha,\sigma) \rightsquigarrow (\beta,\sigma')$ *implies* $(f(\alpha),\sigma) \rightsquigarrow (f(\beta),\sigma')$*;*
- *If* $(f(\alpha),\sigma) \rightsquigarrow (\beta',\sigma')$ *then either* α *is a GITree-value, or there exists* β *such that* $(\alpha,\sigma) \rightsquigarrow (\beta,\sigma')$ *and* $\triangleright(f(\beta) = \beta')$*.*

Finally, as usual in Iris, the program logic satisfies an adequacy property, which allows one to relate propositions proved in the logic to the actual semantics:

Theorem 3.2.3. *Let α be an interaction tree and σ be a state such that*

$$\mathsf{has\_state}(\sigma) \vdash \mathit{wp}\,\alpha\,\{\Phi\}$$

is derivable for some meta-level predicate Φ (containing only intuitionistic logic connectives). Then for any β and σ' such that $(\alpha,\sigma) \rightsquigarrow^* (\beta,\sigma')$, one of the following two things hold:

- (adequacy) *either $\beta \in \mathbf{IT}^v$, and $\Phi(\beta)$ holds in the meta-logic;*
- (safety) *or there are β_1 and σ_1 such that $(\beta,\sigma') \rightsquigarrow (\beta_1,\sigma_1)$*

In particular, safety implies that $\beta \neq \mathsf{Err}(e)$ for any error $e \in \mathtt{Error}$.

The role of meta-logic is played by the Coq system; thus, the adequacy theorem allows us to relate proofs inside the program logic (Iris) to the proofs on the level of Coq. This aspect is important in Iris and GITrees in general, but it is orthogonal to the work that we present in this paper. See [49] for more details.

3.3 Context-Dependent Reification

In this section we extend reification to handle context-dependent effects, using a language $\lambda_{\mathsf{callcc}}$ with `call/cc` as a concrete example. In Section 3.3.1 we present $\lambda_{\mathsf{callcc}}$'s syntax and operational semantics (in the usual style with evaluation contexts). We then show why the current GITrees framework cannot be used as a denotational

$$
\begin{array}{lllll}
\text{types} & \mathit{Ty} \ni \tau & ::= & \mathbb{N} \mid \tau_1 \to \tau_2 \mid \mathit{cont}(\tau) \\
\text{expressions} & \mathit{Expr} \ni e & ::= & v \mid x \in \mathit{Var} \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
 & & & \mid \mathsf{call/cc}\ (\mathsf{x}.\ e) \mid \mathsf{throw}\ e_1\ \mathsf{to}\ e_2 \\
\text{values} & \mathit{Val} \ni v & ::= & n \mid \mathsf{rec}\, f(x) = e \mid \mathsf{cont}\ K \\
\text{eval. cont.} & \mathit{Ectx} \ni K & ::= & \square \mid \mathsf{if}\ K\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid K\ v \mid e\ K \mid e \oplus K \mid K \oplus v \\
 & & & \mid \mathsf{throw}\ K\ \mathsf{to}\ e \mid \mathsf{throw}\ v\ \mathsf{to}\ K
\end{array}
$$

$$
\mathsf{call/cc}\ (\mathsf{x}.\ e) \mapsto_K e[\mathsf{cont}\ K/x] \qquad K[\mathsf{throw}\ v\ \mathsf{to}\ \mathsf{cont}\ K'] \mapsto K'[v] \qquad \frac{e_1 \mapsto_K e_2}{K[e_1] \mapsto K[e_2]}
$$

$$
\frac{\Gamma, x : \mathit{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \mathsf{call/cc}\ (\mathsf{x}.\ e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathit{cont}(\tau)}{\Gamma \vdash \mathsf{throw}\ e_1\ \mathsf{to}\ e_2 : \tau'}
$$

Figure 3.6: Syntax and fragments of type system and operational semantics of $\lambda_{\mathsf{callcc}}$.

model for $\lambda_{\mathsf{callcc}}$ directly. In Section 3.3.2 we introduce our generalization of reification for context-dependent effects and corresponding extensions to the GITrees program logic. In Section 3.3.3 we demonstrate that our extension works as intended: we give a denotational semantics for $\lambda_{\mathsf{callcc}}$, and we show how the general program logic for GITrees specializes to a logic for reasoning about `call/cc`. We prove soundness and computational adequacy of denotational semantics using a logical relation defined within our program logic.

3.3.1 Operational Semantics and Type System for $\lambda_{\mathsf{callcc}}$

By $\lambda_{\mathsf{callcc}}$ we denote a simply-typed λ-calculus with natural numbers, recursive functions, and `call/cc`. The relevant pieces of syntax, the type system and the operational semantics are given in Figure 3.6. The type system includes natural numbers, function types, and the type $\mathit{cont}(\tau)$ of continuations. The $\mathsf{call/cc}\ (\mathsf{x}.\ e)$ expression takes the current evaluation context and binds it to x in e. The $\mathsf{throw}\ e\ \mathsf{to}\ e'$ expression evaluates passes the first argument (value) to the second argument (continuation, represented as an evaluation context).

The operational semantics of $\lambda_{\mathsf{callcc}}$ is separated into two layers. The first layer consists of local reductions of primitive expressions ($e \mapsto_K e'$), and the second layer lifts local reductions to reductions among complete programs ($K[e] \mapsto K'[e']$). The local reductions are parameterized by an evaluation context K, which allows $\mathsf{call/cc}\ (\mathsf{x}.\ e)$ to capture the evaluation context. Let us now consider what happens if we try to give a direct-style denotational semantics of $\lambda_{\mathsf{callcc}}$ into GITrees. By direct we mean that we wish to give a direct interpretation of types and expressions, rather than going through a global CPS conversion. To define the semantics, we first need to provide an effect signature, state, and reifiers for each effect, and then we can define the interpretation of the expressions of the language.

The effect signature, shown in Figure 3.7, contains two effects `callcc` and `throw`. Since `call/cc` binds a continuation, it is natural to let the input arity for `callcc` be

$$Ins_{\texttt{callcc}}(X) \triangleq ((\blacktriangleright X \to \blacktriangleright X) \to \blacktriangleright X) \quad Outs_{\texttt{callcc}}(X) \triangleq \blacktriangleright X$$
$$Ins_{\texttt{throw}}(X) \triangleq \blacktriangleright X \times \blacktriangleright(X \to X) \quad Outs_{\texttt{throw}}(X) \triangleq 0$$
$$\text{Callcc}(f) \triangleq \mathsf{Vis}_{\texttt{callcc}}(f, \mathsf{id}) \quad \text{Throw}(e, f) \triangleq \mathsf{Vis}_{\texttt{throw}}(e, f, \lambda x.\, \mathsf{abort}\ x)$$

Figure 3.7: Signatures and opertaions on GITrees with `call/cc`.

a callback $(\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}) \to \blacktriangleright\mathbf{IT}$. The output arity is simply $\blacktriangleright\mathbf{IT}$.

The input arity for `throw` signifies that `throw` takes as input an expression and a continuation, which are represented respectively as $\blacktriangleright\mathbf{IT}$ and $\blacktriangleright(\mathbf{IT} \to \mathbf{IT})$. The output arity of `throw` is simply the empty type 0, because `throw` never returns.

Note that the input types of `callcc` and `throw` have slightly different arities. However, we can always transform $f : (\blacktriangleright X \to \blacktriangleright X)$ into an element of type $\blacktriangleright(X \to X)$ by performing a silent step in the function's body: $f' \triangleq \mathsf{next}(\lambda x.\, \mathsf{Tau}(f(\mathsf{next}(x))))$. And we can always transform $f : \blacktriangleright(X \to X)$ into an element of type $\blacktriangleright X \to \blacktriangleright X$ using the applicative structure of the later modality.

For convenience, we will use the abbreviations $\text{Callcc}(f)$ and $\text{Throw}(e)$, defined in Figure 3.7, for representing denotations of `throw` and `call/cc` as effects in GITrees.

To complete all the ingredients for the denotational semantics, we need reifiers for the `callcc` and `throw` effects. Given our operational understanding of continuations, the natural choice for the local state type *State* is $\mathbf{1}$ (since we do not have any state). However, the current reifier signature (Figure 3.4) poses a problem. Reifiers, as they are now, cannot access their current continuation, which is essential for both effects. $\text{Callcc}(f)$ needs to pass the current continuation to f, while Throw must redirect control to a provided continuation instead of returning normally. The current reifiers lacks this capability, and in the next subsection we show how to generalize the notion of reification to context-dependent effects.

3.3.2 Context-dependent Reifiers

This section presents our extension to context-dependent reification, and the limitations it imposes on the program logic. In order to allow reifiers to manage continuations, we change the type of reifiers to accept continuations as an extra parameter, as shown in Figure 3.8. Continuations for a given effect are functions from the effect's outputs to GITrees: $Outs_i(\mathbf{IT}_E) \to \blacktriangleright\mathbf{IT}_E$. Given a set of context-dependent reifiers, we define a context-dependent `reify` function, also shown in Figure 3.8. As before, `reify` dispatches to the correct individual reifier for the effect. Note that now it is the user's responsibility to pass the output of an effect to the given continuation if the control flow is not supposed to be interrupted. For example, since the evaluation of a call/cc (x. e) expression does not modify the control flow itself, but simply passes the

$$r : \prod_{\mathtt{i} \in E} \mathit{Ins}_{\mathtt{i}}(\mathbf{IT}_E) \times \mathit{State} \times (\mathit{Outs}_{\mathtt{i}}(\mathbf{IT}_E) \to \blacktriangleright\mathbf{IT}_E) \to \mathit{option}(\blacktriangleright\mathbf{IT}_E \times \mathit{State})$$

$$\frac{r_i(x,\sigma,\kappa) = (\beta,\sigma')}{\mathsf{reify}(\mathsf{Vis}_i(x,\kappa),\sigma) = (\mathsf{Tau}(\beta),\sigma')} \qquad \frac{r_i(x,\sigma,\kappa) = \mathsf{None}}{\mathsf{reify}(\mathsf{Vis}_i(x,\kappa),\sigma) = (\mathsf{Err}(\mathit{RunTime}),\sigma)}$$

Figure 3.8: Type of context-dependent reifiers and the context-dependent reify function

current continuation to its body, the context-dependent reifier for `callcc` is simply $r_{\mathtt{callcc}}(x,\sigma,\kappa) = (\kappa\ (x\ \kappa),\sigma)$.

Before we move on to discussing the consequences of this for program logic, we would like to note that our treatment of continuation (with top-level reifiers dispatching them) parallels Cartwright and Felleisen's "extensible direct models" [28], which also aimed to support extensible denotational semantics in classical domain theory. We discuss this more in Section 3.6.

Program logic for GITrees in the presence of context-dependent reifiers. To reflect the generalization to context-dependent reifiers in the program logic, we replace the proof rule WP-REIFY by WP-REIFY-CTX-DEP, shown in Figure 3.9. This is, however, not the only change we need to make. In the presence of context-dependent effects, WP-HOM is not sound! (A similar observation was also made by [130] in their development of a program logic for `call/cc`.) The reason is that context-dependent reification invalidates Lemma 3.2.2. Now, since WP-HOM is not sound anymore, one might expect that we need to adapt all the other program logic rules to include a homomorphism similarly to how the rules of [130] were adapted to include an evaluation context. However, this is not necessary, because our program logic is defined on *denotations* on which we have a non-trivial equational theory, which can be used to reason about 'pure' GITrees. Only for effectful operations, the proof rules will now have to include a surrounding homomorphism. E.g., WP-WRITE from [49] is generalized to WP-WRITE-CTX-DEP, and considers ambient homomorphisms explicitly.

Our context-dependent reification extension, though simple, allows us to build sound and adequate denotational models for languages with control-flow operators, including $\lambda_{\mathsf{callcc}}$ (shown in the next subsection). Moreover, our extension is conservative, and we recover previous case studies (computational adequacy of $\lambda_{\mathsf{rec,io}}$ and type safety for $\lambda_{\multimap,\mathsf{ref}}$ [49]) with minimal modifications; see the accompanying Coq formalization.

3.3.3 Denotational Semantics of $\lambda_{\mathsf{callcc}}$

In this section we show that context-dependent reifiers are sufficient for providing a sound and adequate semantic model of $\lambda_{\mathsf{callcc}}$. We define context-dependent reifiers for `callcc` and `throw`, then prove that this gives a sound interpretation w.r.t. operational semantics. To show adequacy, we define a logical relation, which relates the

$$\textsc{WP-WRITE}\quad \frac{\mathsf{heap\_ctx} \quad \triangleright \ell \mapsto \alpha \qquad \triangleright(\ell \mapsto \beta \mathrel{-\!\!*} \mathsf{wp}\,\mathsf{Ret}()\,\{\Phi\})}{\mathsf{wp}\,\mathsf{Write}(\ell,\beta)\,\{\Phi\}}$$

$$\textsc{WP-WRITE-CTX-DEP}\quad \frac{\kappa \in \mathit{Hom} \qquad \mathsf{heap\_ctx} \quad \triangleright \ell \mapsto \alpha \qquad \triangleright(\ell \mapsto \beta \mathrel{-\!\!*} \mathsf{wp}\,\kappa\,(\mathsf{Ret}())\,\{\Phi\})}{\mathsf{wp}\,\kappa\,(\mathsf{Write}(\ell,\beta))\,\{\Phi\}}$$

$$\textsc{WP-REIFY-CTX-DEP}\quad \frac{\mathsf{has\_state}(\sigma) \qquad r_i(x,\sigma,k) = (\mathsf{next}(\beta),\sigma') \qquad \triangleright\left(\mathsf{has\_state}(\sigma') \mathrel{-\!\!*} \mathsf{wp}\,\beta\,\{\Phi\}\right)}{\mathsf{wp}\,\mathsf{Vis}_i(x,k)\,\{\Phi\}}$$

Figure 3.9: Program logic in the presence of context-dependent reifiers.

$$\begin{aligned}
&\mathbf{E}[\![x]\!]_\rho = \rho(x)\\
&\mathbf{E}[\![\mathsf{call/cc}\ (\mathsf{x}.\ e)]\!]_\rho = \mathsf{Callcc}(\lambda(f : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}).\,\mathbf{E}[\![e]\!]_{\rho[x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\,\mathsf{Tau}(f(\mathsf{next}(y)))))]})\\
&\mathbf{E}[\![\mathsf{throw}\ e_1\ \mathsf{to}\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda x.\,\mathsf{get_fun}(\mathbf{E}[\![e_2]\!]_\rho, \lambda f.\,\mathrm{Throw}(x,f)))\\
&\mathbf{V}[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\,\mathsf{Tau}(\mathbf{K}[\![K]\!]_\rho\ (\blacktriangleright\bullet)\ \mathsf{next}(x))))\\
&\mathbf{K}[\![\mathsf{throw}\ K\ \mathsf{to}\ e]\!]_\rho = \lambda x.\,\mathsf{get_val}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda y.\,\mathsf{get_fun}(\mathbf{E}[\![e]\!]_\rho, \lambda f.\,\mathrm{Throw}(y,f)))\\
&\mathbf{K}[\![\mathsf{throw}\ v\ \mathsf{to}\ K]\!]_\rho = \lambda x.\,\mathsf{get_val}(\mathbf{V}[\![v]\!]_\rho, \lambda y.\,\mathsf{get_fun}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda f.\,\mathrm{Throw}(y,f)))
\end{aligned}$$

Figure 3.10: Denotational semantics of $\lambda_{\mathsf{callcc}}$ (selected clauses).

denotational and operational semantics. The logical relation is defined in the (updated) program logic for GITrees (following the approach in [49]), and validates the utility of the program logic.

Interpretation of $\lambda_{\mathsf{callcc}}$. The denotational semantics of $\lambda_{\mathsf{callcc}}$ is shown in Figure 3.10 (selected clauses only; see Coq formalization for the complete definition). The interpretation is split into three parts: $\mathbf{E}[\![-]\!]$ for expressions, $\mathbf{V}[\![-]\!]$ for values, and $\mathbf{K}[\![-]\!]$ for contexts. For the interpretation of $\mathsf{throw}\ e_1\ \mathsf{to}\ e_2$, the left-to-right evaluation order is enforced by the functions `get_val` and `get_fun`. They first evaluate their argument to a GITree value, and then pass it on (c.f. Figure 3.3).

The context-dependent reifiers for the effects `callcc` and `throw` are defined as follows:

$$r_{\texttt{callcc}}(f,(),\kappa) = (\kappa\ (f\ \kappa),()) \qquad r_{\texttt{throw}}((\alpha,f),(),\kappa) = (f\ \alpha,())$$

To show that the denotational semantics is sound, we need the following lemma that shows that interpretations of expressions in evaluation contexts are decomposed into applications of homomorphisms.

Lemma 3.3.1. *For any context K and an environment ρ, we have $\mathbf{K}[\![K]\!]_\rho \in Hom$. For any context K, expression e, and an environment ρ, $\mathbf{E}[\![K[e]]\!]_\rho = \mathbf{K}[\![K]\!]_\rho\,(\mathbf{E}[\![e]\!]_\rho)$.*

With these results at hand, we can show soundness of our interpretation:

Lemma 3.3.2. *Soundness. Suppose $e_1 \to e_2$. Then $(\mathbf{E}[\![e_1]\!]_\rho, ()) \rightsquigarrow^* (\mathbf{E}[\![e_2]\!]_\rho, ())$, where $() : \mathbf{1}$ is the unique element of the unit type, representing the (lack of) state.*

Program logic for $\lambda_{\mathsf{callcc}}$. We now specialize the general program logic rule WP-REIFY-CTX-DEP using the reifiers for `callcc` and `throw` to obtain the following program logic rules:

$$\frac{\kappa \in Hom \quad \mathsf{has\_state}(\sigma) \quad \triangleright(\mathsf{has\_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\, f\, x\, \{\Phi\})}{\mathsf{wp}\,\kappa\,(\mathrm{Throw}(\mathsf{next}(x), \mathsf{next}(f)))\,\{\Phi\}}\ \text{WP-THROW}$$

$$\frac{\kappa \in Hom \quad \mathsf{has\_state}(\sigma) \quad \triangleright(\mathsf{has\_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\,\kappa\,(f\,\kappa)\,\{\Phi\})}{\mathsf{wp}\,\kappa\,(\mathrm{Callcc}(\mathsf{next} \circ f))\,\{\Phi\}}\ \text{WP-CALLCC}$$

where κ is a homomorphism representing the current evaluation context on the level of GITrees. The reader may wonder why these rules include the $\mathsf{has\_state}(\sigma)$ predicates, since it is just 'threaded around'. The reason is that these rules also apply when there are other effects around and the state is composed of different substates for different effects, cf. the discussion of modularity in Section 3.2.

Adequacy and logical relation. Having established soundness, we now turn our attention to *adequacy*, which is usually much more complicated to prove.

Lemma 3.3.3. *Adequacy. Suppose that $\emptyset \vdash e : \mathbb{N}$ and $(\mathbf{E}[\![e]\!]_\emptyset, \sigma_1) \rightsquigarrow^* (\mathsf{Ret}(n), \sigma_2)$, for a natural number n. Then $e \mapsto^* n$.*

To prove Lemma 3.3.3, we define a logical relation between syntax ($\lambda_{\mathsf{callcc}}$ programs) and semantics (GITrees denotations) using the program logic from Figure 3.11. To handle control effects, we use a *biorthogonal* logical relation [97], adapted from [130] for adequacy, following the Iris approach [133].

The core observational refinement $\mathcal{O}(\alpha, e)$ ensures that if α reduces to a GITree value $\mathbf{IT}^v$, then this value is a natural number, and e also reduces to the same number. The evaluation context relation $\mathcal{K}(P)(\kappa, K)$ relates homomorphisms and evaluation contexts when they map related arguments to expressions satisfying $\mathcal{O}$. The expression relation $\mathcal{E}(P)(\alpha, e)$ connects related $\mathbf{IT}$'s and expressions in related evaluation contexts. Types are inductively interpreted: functions relate if they map related arguments to related results, and continuations relate via the context relation. For open terms, the validity judgment $\Gamma \vDash e : \tau$ uses closing substitutions, with $e[\gamma]$ denoting applying a substitution γ to e.

$$\mathcal{O}(\alpha, e) \triangleq \mathsf{has\_state}(()) \mathrel{-\!\!*} \mathsf{wp}\, \alpha \left\{\beta.\exists v.\, (e \mapsto^* v) * [\![\mathbb{N}]\!]\,(\beta, v) * \mathsf{has\_state}(())\right\} \qquad \boxed{\mathcal{O} : \mathrm{ERel}}$$

$$\mathcal{K}(R)(\kappa, K) \triangleq \Box\, \forall(\beta, v).\, R(\beta, v) \mathrel{-\!\!*} \mathcal{O}(\kappa\, \beta, K[v]) \qquad \boxed{\mathcal{K} : \mathrm{VRel} \to \mathrm{CRel}}$$

$$\mathcal{E}(R)(\alpha, e) \triangleq \forall(\kappa, K).\, \mathcal{K}(R)(\kappa, K) \mathrel{-\!\!*} \mathcal{O}(\kappa\, \alpha, K[e]) \qquad \boxed{\mathcal{E} : \mathrm{VRel} \to \mathrm{ERel}}$$

$$[\![\mathbb{N}]\!]\,(\alpha, v) \triangleq \exists n : \mathbb{N}.\, \alpha = \mathsf{Ret}(n) \wedge v = n \qquad \boxed{[\![\tau]\!] : \mathrm{VRel}}$$

$$[\![\tau_1 \to \tau_2]\!]\,(\beta, v) \triangleq \exists f.\, \beta = \mathsf{Fun}(f) \wedge \Box\, \forall(\alpha, v').\, [\![\tau_1]\!]\,(\alpha, v') \mathrel{-\!\!*} \mathcal{E}([\![\tau_2]\!])(\mathsf{Fun}(f) \bullet \alpha, v\, v')$$

$$[\![\mathit{cont}(\tau)]\!]\,(\beta, v) \triangleq \exists \kappa\, K.\, \beta = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tick}(\kappa\, x))) \wedge v = \mathsf{cont}\ K \wedge \mathcal{K}([\![\tau]\!])(\kappa, K)$$

$$[\![\Gamma]\!]\,(\rho, \gamma) \triangleq \forall(x : \tau) \in \Gamma.\, [\![\tau]\!](\rho\, x, \gamma\, x)$$

$$\Gamma \vDash e : \tau \triangleq \Box\, \forall(\rho, \gamma).\, [\![\Gamma]\!]\,(\rho, \gamma) \mathrel{-\!\!*} \mathcal{E}([\![\tau]\!])(\mathbf{E}\,[\![e]\!]_\rho, e[\gamma])$$

$$\mathrm{CRel} \triangleq \mathit{Hom} \times \mathit{Ectx} \to \mathit{iProp}$$
$$\mathrm{VRel} \triangleq \mathbf{IT}^v \times \mathit{Val} \to \mathit{iProp}$$
$$\mathrm{ERel} \triangleq \mathbf{IT} \times \mathit{Expr} \to \mathit{iProp}$$

Figure 3.11: Logical relation for $\lambda_{\mathsf{callcc}}$.

The proof of adequacy relies on the fact that the interpretation of evaluation contexts are homomorphisms, which allows us to use a limited version of the bind rule:

Lemma 3.3.4. *Limited bind rule. If $\mathcal{E}(P)(\alpha, e)$ and $\mathcal{K}(P)(\kappa, K)$, then $\mathcal{O}(\kappa\, \alpha, K[e])$.*

With this in mind we show the fundamental lemma, stating that every well-typed expression is related to its own interpretation:

Lemma 3.3.5. *Fundamental lemma. Let $\Gamma \vdash e : \tau$ then $\Gamma \vDash e : \tau$.*

Computational adequacy now follows easily from the fundamental lemma.

of Lemma 3.3.3. By Lemma 3.3.5, we have that $\emptyset \vdash e : \mathbb{N}$ implies that $\emptyset \vDash e : \mathbb{N}$. Now, the statement follows from Theorem 3.2.3 and the assumption that $\mathbf{E}\,[\![e]\!]_\emptyset, \sigma_1 \rightsquigarrow^* \mathsf{Ret}\ n, \sigma_2$. □

3.4 Modeling Delimited Continuations

In this section we scale our approach to delimited continuations, which is a challenging example of context-dependent effects. We provide a denotational semantics for a programming language λ_{delim} with `shift`/`reset`, and prove its soundness and adequacy relative to an abstract machine semantics [18]. The semantics and proofs are more complex than for $\lambda_{\mathsf{callcc}}$ due to the nature of delimited continuations and associated type system. To the best of our knowledge, this represents the first formalized sound and adequate direct-style denotational semantics for delimited continuations.

$\boxed{\begin{array}{l}\Gamma;\alpha \vdash e : \tau;\beta \\ \Gamma \vdash_{\mathsf{pure}} e : \tau\end{array}}$

types	$Ty \ni \tau,\sigma,\alpha,\beta,\delta,\gamma$	$::=$	$\mathbb{N} \mid \tau/\alpha \to \sigma/\beta \mid \mathit{cont}(\tau,\alpha)$
expressions	$\mathit{Expr} \ni e$	$::=$	$v \mid x \mid e_1\ e_2 \mid e_1 \oplus e_2$
			$\mid \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \mid \mathcal{S}\ \mathsf{x}.\ e \mid \mathcal{D}\ \mathsf{e} \mid e_1\ @\ e_2$
values	$\mathit{Val} \ni v$	$::=$	$n \mid \mathsf{rec}\, f(x) = e \mid \mathsf{cont}\ K$
eval. contexts	$\mathit{Ectx} \ni K$	$::=$	$\square \mid K[\mathsf{if}\ \square\ \mathsf{then}\, e_1\, \mathsf{else}\, e_2] \mid K[v\ \square] \mid K[\square\ e]$
			$\mid K[e \oplus \square] \mid K[\square \oplus v] \mid K[\square\ @\ v] \mid K[e\ @\ \square]$

$$\frac{\Gamma \vdash_{\mathsf{pure}} e : \tau}{\Gamma;\alpha \vdash e : \tau;\alpha} \qquad \frac{\Gamma, x : \mathsf{cont}\ (\tau,\alpha);\sigma \vdash e : \sigma;\beta}{\Gamma;\alpha \vdash \mathcal{S}\ \mathsf{x}.\ e : \tau;\beta} \qquad \frac{\Gamma;\tau \vdash e : \tau;\sigma}{\Gamma \vdash_{\mathsf{pure}} \mathcal{D}\ \mathsf{e} : \sigma} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathsf{pure}} x : \tau}$$

$$\frac{\Gamma, f : \sigma/\alpha \to \tau/\beta, x : \sigma;\alpha \vdash e : \tau;\beta}{\Gamma \vdash_{\mathsf{pure}} \mathsf{rec}\, f(x) = e : \sigma/\alpha \to \tau/\beta} \qquad \frac{\Gamma;\gamma \vdash e_1 : \sigma/\alpha \to \tau/\beta;\delta \quad \Gamma;\beta \vdash e_2 : \sigma;\gamma}{\Gamma;\alpha \vdash e_1\ e_2 : \tau;\delta}$$

$$\frac{\Gamma;\beta \vdash e_1 : \mathbb{N};\alpha \quad \Gamma;\sigma \vdash e_2 : \tau;\beta \quad \Gamma;\sigma \vdash e_3 : \tau;\beta}{\Gamma;\sigma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau;\alpha} \qquad \frac{}{\Gamma \vdash_{\mathsf{pure}} n : \mathbb{N}}$$

$$\frac{\Gamma;\alpha \vdash e_1 : \mathbb{N};\beta \quad \Gamma;\beta \vdash e_2 : \mathbb{N};\sigma}{\Gamma;\alpha \vdash e_1 \oplus e_2 : \mathbb{N};\sigma} \qquad \frac{\Gamma;\sigma \vdash e_1 : \mathsf{cont}\ (\tau,\alpha);\delta \quad \Gamma;\delta \vdash e_2 : \tau;\beta}{\Gamma;\sigma \vdash e_1\ @\ e_2 : \alpha;\beta}$$

Figure 3.12: Syntax and typing rules of λ_{delim}.

$$\begin{aligned}
\langle e \rangle_{\mathsf{term}} &\mapsto \langle e,\ \square,\ [] \rangle_{\mathsf{eval}} \\
\langle K :: mk,\ v \rangle_{\mathsf{mcont}} &\mapsto \langle K,\ v,\ mk \rangle_{\mathsf{cont}} \\
\langle [],\ v \rangle_{\mathsf{mcont}} &\mapsto \langle v \rangle_{\mathsf{ret}} \\
\langle \square,\ v,\ mk \rangle_{\mathsf{cont}} &\mapsto \langle mk,\ v \rangle_{\mathsf{mcont}} \\
\langle K[\square\ @\ v],\ \mathsf{cont}\ K',\ mk \rangle_{\mathsf{cont}} &\mapsto \langle K',\ v,\ K :: mk \rangle_{\mathsf{cont}} \\
\langle K[e\ @\ \square],\ v,\ mk \rangle_{\mathsf{cont}} &\mapsto \langle e,\ K[\square\ @\ v],\ mk \rangle_{\mathsf{eval}} \\
\langle v,\ K,\ mk \rangle_{\mathsf{eval}} &\mapsto \langle K,\ v,\ mk \rangle_{\mathsf{cont}} \\
\langle e_0\ @\ e_1,\ K,\ mk \rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[e_0\ @\ \square],\ mk \rangle_{\mathsf{eval}} \\
\langle \mathcal{D}\ \mathsf{e},\ K,\ mk \rangle_{\mathsf{eval}} &\mapsto \langle e,\ \square,\ K :: mk \rangle_{\mathsf{eval}} \\
\langle \mathcal{S}\ \mathsf{k}.\ e,\ K,\ mk \rangle_{\mathsf{eval}} &\mapsto \langle e[K/k],\ \square,\ mk \rangle_{\mathsf{eval}}
\end{aligned}$$

metacontinuations:
$\mathit{Mcont} \ni mk ::= [] \mid K :: mk$
abstract machine config.:
$\mathit{Config} ::= \langle e,\ K,\ mk \rangle_{\mathsf{eval}} \mid \langle K,\ v,\ mk \rangle_{\mathsf{cont}} \mid \langle mk,\ v \rangle_{\mathsf{mcont}} \mid \langle e \rangle_{\mathsf{term}} \mid \langle v \rangle_{\mathsf{ret}}$

Figure 3.13: Operational semantics of λ_{delim} (excerpt).

3.4.1 Syntax and Operational Semantics of λ_{delim}

The syntax and the type system of λ_{delim} is given in Figure 3.12. It is similar to $\lambda_{\mathsf{callcc}}$, but instead of $\mathsf{call/cc}\ (-.\ -)$ there are operators $\mathcal{D}\ \mathsf{e}$ (delimit the current evaluation context, also known as `reset`) and $\mathcal{S}\ \mathsf{x}.\ e$ (grab the current delimited continuation, and bind it to x in e, also known as `shift`).

The type system follows Danvy and Filinski [32], extending simply-typed λ-calculus with answer types α, β. The main typing judgment $\Gamma;\alpha \vdash e : \tau;\beta$ means:

under the typing context Γ, expression e can be plugged into a context expecting a value of type τ and producing a value of type α; in that case the resulting program will have the type β. You can think of it a computation $\Gamma \to (\tau \to \alpha) \to \beta$ under the CPS translation. Thus, the type of the (delimited) continuation corresponds to $\tau \to \alpha$, while the type of the overall expression is β. The pure typing judgment $\Gamma \vdash_{\mathsf{pure}} e : \tau$ indicates e does not depend on the surrounded context, and is context-independent for any answer types. Expressions can change their context's answer type, as seen in the $\mathcal{S}\ \mathsf{x}.\ e$ typing rule.

For example, suppose we extend the type system with booleans, $\mathbb{B}$, and add a primitive function isprime that does not modify answer types. That is $\emptyset; \alpha \vdash \mathsf{isprime} : \mathbb{N}/\beta \to \mathbb{B}/\beta; \alpha$. The expression $\mathcal{D}\ ((\mathsf{rec}\,\mathsf{f}(\mathsf{x}) = \mathsf{isprime}\ (\mathcal{S}\ \mathsf{k}.\ \mathsf{x} - 1))\ 2)$ is well-typed in this type system as a $\mathbb{N}$, even though it changes the answer type from $\mathbb{B}$ to $\mathbb{N}$.

Answer types appear in both judgments and type constructors. Continuation type $\mathsf{cont}\ (\tau, \alpha)$ represents contexts expecting something of the type τ and producing something of the type α. Function type $\sigma/\alpha \to \tau/\beta$, in addition to the input type σ and the output type τ, record the typing of the surrounding context at the point of the function call. See [32] for details.

The operational semantics for λ_{delim} uses a CEK machine, following [17, 18, 44]. Selected reduction rules appear in Figure 3.13 (see Coq formalization for the full set of rules). The abstract machine operates on various *configurations*, which can be of several forms. The first one is the initial configuration $\langle e \rangle_{\mathsf{term}}$, which is just a starting state for evaluating expressions. Similarly, there is a terminal configuration $\langle v \rangle_{\mathsf{ret}}$ signifying that the program has terminated with the value v.

From the initial configuration, we go on to $\langle e,\ K,\ mk \rangle_{\mathsf{eval}}$, which signifies that we are evaluating an expression e inside the current delimited context K, with the metacontinuation mk (a stack of continuations based on different delimiters). It is this configuration type which takes care of delimited control operations. The $\mathcal{D}$ operator saves the current continuation on top of the metacontinuation, limiting the scope of $\mathcal{S}\ \mathsf{x}.\ e$. The $\mathcal{S}\ \mathsf{x}.\ e$ operation behaves similarly to $\mathsf{call/cc}\ (\mathsf{x}.\ e)$, except that it prevents later control operators from capturing its evaluation context.

The last two configuration types are for dealing with continuations and metacontinuations. A configuration $\langle K,\ v,\ mk \rangle_{\mathsf{cont}}$ signifies that we are trying to plug in the value v into the context K, with the metacontinuation mk. A configuration $\langle mk,\ v \rangle_{\mathsf{mcont}}$ signifies that we are done with the current continuation (ending with the value v), but we still have to unwind the continuation stack mk.

3.4.2 Denotational Semantics of λ_{delim}

Our model represents delimited continuations with effects mimicking an abstract machine, operating on semantic rather than syntactic components. The effect signature and reifiers (Figure 3.14) define a state with a stack of continuations, manipulated explicitly. The effect signature $E_{\lambda_{\mathsf{delim}}}$ includes four operators: $\{\mathtt{reset}, \mathtt{shift}, \mathtt{pop}, \mathtt{appcont}\}$. The signature of `reset` simply tells us that the corresponding effect does not directly modify its argument. The auxiliary effect `pop`,

$$
\begin{aligned}
Ins_{\texttt{reset}}(X) &\triangleq \blacktriangleright X & \qquad Ins_{\texttt{shift}}(X) &\triangleq (\blacktriangleright X \to \blacktriangleright X) \to \blacktriangleright X \\
Ins_{\texttt{pop}}(X) &\triangleq \blacktriangleright X & Ins_{\texttt{appcont}}(X) &\triangleq \blacktriangleright X \times \blacktriangleright (X \to X) \\
Outs_{\texttt{reset}}(X) &\triangleq \blacktriangleright X & Outs_{\texttt{shift}}(X) &\triangleq \blacktriangleright X \\
Outs_{\texttt{pop}}(X) &\triangleq 0 & Outs_{\texttt{appcont}}(X) &\triangleq \blacktriangleright X \\
r_{\texttt{reset}}(e, \sigma, \kappa) &= (e, \kappa :: \sigma) & r_{\texttt{shift}}(f, \sigma, \kappa) &= (f\ \kappa, \sigma) \\
r_{\texttt{pop}}(e, [], \_) &= (e, []) & r_{\texttt{pop}}(e, \kappa :: \sigma, \_) &= (\kappa\ e, \sigma) \\
r_{\texttt{appcont}}((e, \kappa), \sigma, \kappa') &= (\kappa\ e, \kappa' :: \sigma) & \mathbf{P}(\beta) &\triangleq \mathsf{get\_val}(\beta, \mathsf{Pop}) \\
\mathsf{Reset}(e) &\triangleq \mathsf{Vis}_{\texttt{reset}}(e, \mathsf{id}) & \mathsf{Shift}(f) &\triangleq \mathsf{Vis}_{\texttt{shift}}(f, \mathsf{id}) \\
\mathsf{Appcont}(e, f) &\triangleq \mathsf{Vis}_{\texttt{appcont}}((e, f), \mathsf{id}) & \mathsf{Pop}(e) &\triangleq \mathsf{Vis}_{\texttt{pop}}(e, \lambda x.\, \mathsf{abort}\, x)
\end{aligned}
$$

Figure 3.14: Effects for λ_{delim}.

which does not have an equivalent in the surface syntax, is used to enforce unwinding of the continuation stack. As the output arity of `pop` signifies, it does not return. We describe the importance of that below. The rest of the signature is more straightforward: `shift` and `appcont` are defined exactly as `callcc` and `throw`. The semantics of these effects, in terms of reification, is more intricate. As we mentioned, the state for reification is $State = List(\blacktriangleright \mathbf{IT} \to \blacktriangleright \mathbf{IT})$.

In comparison with call/cc $(\mathsf{x}.\ e)$, the control operator $\mathcal{S}\ \mathsf{x}.\ e$ does not necessarily continue from the same continuation; hence, the corresponding reifier passes the current continuation to the body, but does not return control back. The reifier for `reset` simply saves the current continuation κ onto the stack σ. It is then the job of the `pop` operation to restore the continuation from the stack. The reifier for `shift` is similar to that of `callcc`, except that it removes the current continuation entirely. The reifier for `appcont`, in comparison with `throw`, does not simply pass control, but also saves the current continuation on the stack. This corresponds to the fact that whenever a delimited continuation is invoked, the result is wrapped in a reset; that is done to prevent the continuation from escaping the delimiter. As part of instantiating GITrees with these effects, we obtain the specialized program logic rules shown in Figure 3.15. We will use those rules later for defining a logical relation between the syntax and the semantics of λ_{delim}. As mentioned above, we will use Pop to unwind the continuation stack and restore the continuation after finishing with a `reset`. This means that we will need to insert explicit calls to Pop in the interpretation of λ_{delim}. For these purposes, we use an abbreviation $\mathbf{P}(\beta)$, which first evaluates β to a value, and then executes the `pop` operation.

The interpretation of λ_{delim} uses this auxiliary function and is given in Figure 3.16. Similarly to the operational semantics, the interpretation is divided into five categories. First, we have $\mathbf{E}[\![-]\!]$ and $\mathbf{V}[\![-]\!]$ for the interpretation of expressions and values, which is what we need for the surface syntax. All of those interpretations return GITrees. Note that in the interpretation of $\mathcal{D}\ -$ we insert explicit calls to $\mathbf{P}$, and similarly in the

WP-SHIFT

$$\frac{\mathsf{has\_state}(\sigma) \quad \triangleright(\mathsf{has\_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\,\beta\,\{\Phi\}) \quad \blacktriangleright\mathbf{P}(f(\blacktriangleright\kappa)) = \mathsf{next}(\beta)}{\mathsf{wp}\,\kappa(\mathsf{Shift}(f))\,\{\Phi\}}$$

WP-RESET

$$\frac{\mathsf{has\_state}(\sigma) \quad \triangleright(\mathsf{has\_state}(\blacktriangleright\kappa :: \sigma) \mathrel{-\!\!*} \mathsf{wp}\,\mathbf{P}(e)\,\{\Phi\})}{\mathsf{wp}\,\kappa(\mathsf{Reset}(\mathsf{next}(e)))\,\{\Phi\}}$$

WP-POP

$$\frac{\mathsf{has\_state}(\sigma) \quad \kappa' = \kappa \text{ if } \sigma = \kappa :: \sigma' \text{ and id otherwise} \quad \triangleright(\mathsf{has\_state}(\mathtt{tail}(\sigma)) \mathrel{-\!\!*} \mathsf{wp}\,\kappa'(v)\,\{\Phi\})}{\mathsf{wp}\,\mathbf{P}(v)\,\{\Phi\}}$$

WP-APPCONT

$$\frac{\mathsf{has\_state}(\sigma) \quad \triangleright(\mathsf{has\_state}(\blacktriangleright\kappa :: \sigma) \mathrel{-\!\!*} \mathsf{wp}\,\beta\,\{\Phi\}) \quad \blacktriangleright\kappa'(e) = \mathsf{next}(\beta)}{\mathsf{wp}\,\kappa(\mathsf{Appcont}(e,\kappa'))\,\{\Phi\}}$$

Figure 3.15: Weakest precondition rules for delimited continuations.

interpretation of continuations.

The other group of interpretations, $\mathbf{K}[\![-]\!]$, $\mathbf{M}[\![-]\!]$ and $\mathbf{S}[\![-]\!]$, are for interpreting continuations, metacontinuations, and other configurations; these are used for showing soundness (preservation of operational semantics by the interpretation). The interpretation $\mathbf{K}[\![-]\!]$ of continuations returns a semantic continuation (a function $\mathbf{IT} \to \mathbf{IT}$). Similarly, the interpretations $\mathbf{M}[\![-]\!]$ (resp. $\mathbf{S}[\![-]\!]$) of metacontinuations (resp. configurations) returns a stack of semantic continuations (resp. a semantic configuration).

We now show that our interpretation is sound w.r.t. the abstract machine semantics. For this we prove lemmas similar to Lemma 3.3.1, and put them to use in the soundness theorem:

Theorem 3.4.1. *Soundness. Let $c_0, c_1 \in$ Config and suppose $c_0 \to c_1$. Then $\mathbf{S}[\![c_0]\!] \rightsquigarrow^* \mathbf{S}[\![c_1]\!]$.*

3.4.3 Logical Relation and Adequacy

We now show that our denotational semantics is adequate with regards to the abstract machine semantics. Specifically, we show the following result:

Theorem 3.4.2. *Adequacy. Suppose $\emptyset; \mathbb{N} \vdash e : \mathbb{N}; \mathbb{N}$ is a well-typed term, and that $(\mathbf{P}(\mathbf{E}[\![e]\!]_\emptyset), []) \rightsquigarrow^* (\mathsf{Ret}(n), \sigma)$ for a natural number n and a metacontinuation σ. Then $\langle e \rangle_{\mathsf{term}} \mapsto^* \langle n \rangle_{\mathsf{ret}}$.*

$$\mathbf{E}\,[\![\mathcal{D}\ \mathsf{e}]\!]_\rho = \mathsf{Reset}(\mathbf{P}(\mathbf{E}\,[\![e]\!]_\rho))$$
$$\mathbf{E}\,[\![\mathcal{S}\ \mathsf{x}.\ e]\!]_\rho = \mathsf{Shift}(\mathbf{P} \circ (\lambda\kappa.\, \mathbf{E}\,[\![e]\!]_{\rho, x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\, \mathsf{Tau}(\kappa(\mathsf{next} y))))}))$$
$$\mathbf{E}\,[\![e_1\ @\ e_2]\!]_\rho = \mathsf{get\_val}(\mathbf{E}\,[\![e_2]\!]_\rho, \lambda x.\, \mathsf{get\_fun}(\mathbf{E}\,[\![e_1]\!]_\rho, \lambda y.\, \mathsf{Appcont}(\mathsf{next}(x), y)))$$
$$\mathbf{V}\,[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tick}(\mathbf{P}(\mathbf{K}\,[\![K]\!]_\rho\ x))))$$
$$\mathbf{K}\,[\![K[\square\ @\ v]]\!]_\rho = \lambda x.\, \mathbf{K}\,[\![K]\!]_\rho\,(\mathbf{E}\,[\![x\ @\ v]\!]_\rho)$$
$$\mathbf{M}\,[\![mk]\!]_\rho = \mathrm{map}(\lambda k.\, \mathbf{P} \circ \mathbf{K}\,[\![k]\!]_\rho) mk$$
$$\mathbf{S}\,[\![\langle e,\ K,\ mk\rangle_{\mathsf{eval}}]\!]_\rho = \Big(\mathbf{P}(\mathbf{E}\,[\![K[e]]\!]_\rho), \mathbf{M}\,[\![mk]\!]_\rho\Big)$$
$$\mathbf{S}\,[\![\langle K,\ v,\ mk\rangle_{\mathsf{cont}}]\!]_\rho = \Big(\mathbf{P}(\mathbf{E}\,[\![K[v]]\!]_\rho), \mathbf{M}\,[\![mk]\!]_\rho\Big)$$
$$\mathbf{S}\,[\![\langle mk,\ v\rangle_{\mathsf{mcont}}]\!]_\rho = \Big(\mathbf{P}(\mathbf{V}\,[\![v]\!]_\rho), \mathbf{M}\,[\![mk]\!]_\rho\Big)$$
$$\mathbf{S}\,[\![\langle e\rangle_{\mathsf{term}}]\!]_\rho = \Big(\mathbf{P}(\mathbf{E}\,[\![e]\!]_\rho), []\Big)$$
$$\mathbf{S}\,[\![\langle v\rangle_{\mathsf{ret}}]\!]_\rho = \Big(\mathbf{V}\,[\![v]\!]_\rho, []\Big)$$

Figure 3.16: Denotational semantics for a calculus with delimited control (selected clauses).

We prove adequacy using a logical relation. It relates expressions to their interpretations and also connects syntactic and semantic configurations. The logical relation is shown in Figure 3.17. It is again a form of biorthogonal logical relation, with the main focus being the observational refinement $\mathcal{O}$: two configurations are related if they reduce to the same natural number. This coincides with what we want to show in Theorem 3.4.2. To facilitate this we then lift $\mathcal{O}$ to the levels of metacontinuations, continuations and expressions. The relation $\mathcal{M}(P)$, where $P : \mathrm{VRel}$ is a relation on values, states that two metacontinuations are related if, whenever we plug in P-related values, the resulting configurations become $\mathcal{O}$-related. Both $\mathcal{M}$ and $\mathcal{O}$ are then used to define the relation between semantic and syntactic continuations. The relation $\mathcal{K}(Q,P)$, where $P,Q : \mathrm{VRel}$ are relations on values, states that two continuations are related if, whenever we plug them into Q-related metacontinuations with P-related values, the resulting configurations become $\mathcal{O}$-related. Finally, we use $\mathcal{K}, \mathcal{M}$, and $\mathcal{O}$ to define the relation between GITrees and λ_{delim} terms. The relation $\mathcal{E}(P,Q,R)$, where $P,Q,R : \mathrm{VRel}$ are relations on values, states that β is related to e if, whenever we plug them into (P,Q)-related continuations and an R-related metacontinuation, the resulting configurations become $\mathcal{O}$-related.

The relations $\mathcal{E}$ and $\mathcal{K}$ are used to give semantics $[\![\tau]\!]$ to types. The idea is that $\mathcal{E}([\![\tau]\!], [\![\alpha]\!], [\![\beta]\!])$ relates terms $\emptyset; \alpha \vdash e : \tau; \beta$ to their semantic counterparts. This is then used, as expected for logical relations, for defining the logical relation for function types and for open terms. The relation $[\![\Gamma]\!]\,(\rho, \gamma)$ relates the semantic environment

$$\begin{aligned}
\text{SynConf} &\triangleq \mathit{Expr} \times \mathit{Ectx} \times \mathit{Mcont} \\
\text{SemConf} &\triangleq \mathbf{IT} \times \mathit{Hom} \times \mathit{List}(\mathit{Hom}) \\
\text{ConfRel} &\triangleq \text{SemConf} \times \text{SynConf} \to \mathit{iProp} \\
\text{MRel} &\triangleq \mathit{list}\ \mathit{Hom} \times \mathit{Mcont} \to \mathit{iProp} \\
\text{CRel} &\triangleq \mathit{Hom} \times \mathit{Ectx} \to \mathit{iProp} \\
\text{ERel} &\triangleq \mathbf{IT} \times \mathit{Expr} \to \mathit{iProp}
\end{aligned}
\qquad
\begin{aligned}
\text{VRel} &\triangleq \mathbf{IT}^v \times \mathit{Val} \to \mathit{iProp} \\
\mathcal{O} &: \text{ConfRel} \\
\mathcal{M} &: \text{VRel} \to \text{MRel} \\
\mathcal{K} &: \text{VRel} \to \text{VRel} \to \text{CRel} \\
\mathcal{E} &: \text{VRel} \to \text{VRel} \to \text{VRel} \to \text{ERel} \\
[\![\tau]\!] &: \text{VRel}
\end{aligned}$$

$$\begin{aligned}
\mathcal{O}\big((\alpha,\kappa,\sigma),(e,K,mk)\big) &\triangleq \mathsf{has_state}(\sigma) \mathrel{-\!\!*} \\
&\quad \mathsf{wp}\mathbf{P}(\kappa\ \alpha)\left\{\beta.\, \exists v.\, (\langle e,\ K,\ mk\rangle_{\mathsf{eval}} \mapsto^* \langle v\rangle_{\mathsf{ret}}) \right. \\
&\quad \left. * (\beta, v) \in [\![\mathbb{N}]\!] * \mathsf{has_state}([])\right\} \\
\mathcal{M}(P)(\sigma,mk) &\triangleq \forall(\alpha,v).\, P(\alpha,v) \mathrel{-\!\!*} \mathcal{O}((\alpha,\iota,\sigma),(v,\square,mk)) \\
\mathcal{K}(Q,P)(\kappa,K) &\triangleq \square\, \forall(\alpha,v).Q(\alpha,v) \mathrel{-\!\!*} \forall(\sigma,mk).\mathcal{M}(P)(\sigma,mk) \mathrel{-\!\!*} \\
&\quad \mathcal{O}((\alpha,\kappa,\sigma),(v,K,mk)) \\
\mathcal{E}(P,Q,R)(\beta,e) &\triangleq \forall(\kappa,K).\mathcal{K}(P,Q)(\kappa,K) \mathrel{-\!\!*} \forall(\sigma,mk).\mathcal{M}(R)(\sigma,mk) \mathrel{-\!\!*} \\
&\quad \mathcal{O}((\beta,\kappa,\sigma),(e,K,mk))
\end{aligned}$$

$$\begin{aligned}
[\![\mathbb{N}]\!](\beta,v) &\triangleq \exists n \in \mathbb{N}.\, \beta = \mathsf{Ret}(n) \wedge v = n \\
[\![\tau/\alpha \to \sigma/\beta]\!](\theta,v) &\triangleq \exists F.\, \theta = \mathsf{Fun}(F) \wedge \square\, \forall(\eta,w).\, [\![\tau]\!](\eta,w) \mathrel{-\!\!*} \\
&\quad \mathcal{E}([\![\sigma]\!],[\![\alpha]\!],[\![\beta]\!])(\theta \bullet \eta, v\ w) \\
[\![\mathit{cont}(\tau,\alpha)]\!](\beta,v) &\triangleq \exists \kappa K.\, \beta = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tick}((\mathbf{P} \circ \kappa)\ x))) \wedge v = \mathsf{cont}\ K \wedge \\
&\quad \mathcal{K}([\![\tau]\!],[\![\alpha]\!])(\kappa,K) \\
[\![\Gamma]\!](\rho,\gamma) &\triangleq \forall(x : \tau \in \Gamma).\, \square\, \forall\Phi.\, \mathcal{E}([\![\tau]\!],\Phi,\Phi)(\rho(x),\gamma(x)) \\
\Gamma \vDash_{\mathsf{pure}} e : \tau &\triangleq \square\, \forall(\rho,\gamma).\, [\![\Gamma]\!](\rho,\gamma) \mathrel{-\!\!*} \forall\Phi.\, \mathcal{E}([\![\tau]\!],\Phi,\Phi)(\mathbf{E}\,[\![e]\!]_\rho, e[\gamma]) \\
\Gamma;\alpha \vDash e : \tau;\beta &\triangleq \square\, \forall(\rho,\gamma).\, [\![\Gamma]\!](\rho,\gamma) \mathrel{-\!\!*} \mathcal{E}([\![\tau]\!],[\![\alpha]\!],[\![\beta]\!])(\mathbf{E}\,[\![e]\!]_\rho, e[\gamma])
\end{aligned}$$

Figure 3.17: Logical relation for λ_{delim}.

$\rho : \mathit{Var} \to \mathbf{IT}$ to the syntactic substitution $\gamma : \mathit{Var} \to \mathit{Expr}$; they are related if they map the same variables to related GITrees/expressions. Then we say that an expression e is semantically valid, $\Gamma;\alpha \vdash e : \tau;\beta$, if its interpretation $\mathbf{E}\,[\![e]\!]_\rho$ is related to $e[\gamma]$ under related substitutions ρ,γ. Note that if we ignore the answer types we can see that the logical relation exhibits a lot of similarities to the logical relation we gave in Section 3.3.3, and follows the same roadmap.

For this logical relation we obtain the fundamental property, which we will use for the proof of adequacy.

Lemma 3.4.3. *Fundamental lemma. Let* $\Gamma;\alpha \vdash e : \tau;\beta$ *then* $\Gamma;\alpha \vDash e : \tau;\beta$*; and if* $\Gamma \vdash_{\mathsf{pure}} e : \tau$ *then* $\Gamma \vDash_{\mathsf{pure}} e : \tau$.

of Theorem 3.4.2. Note that the empty (meta)continuation is related to its denotation: $\mathcal{K}(P,P)(\mathsf{id},\square)$ and $\mathcal{M}(P)([],[])$ hold for any relation P.

$$\begin{array}{lll} \text{types} & \mathit{Ty} \ni \tau & ::= \mathbb{N} \mid \mathbf{1} \mid \tau \to \sigma \mid \mathsf{ref}(\tau) \\ \text{expressions} & \mathit{Expr} \ni e & ::= x \mid () \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid n \mid \lambda x.e \\ & & \quad \mid \ell \mid \mathsf{ref}(e) \mid !e \mid e_1 \leftarrow e_2 \mid \mathsf{embed}\ e \end{array}$$

$$\frac{\emptyset \vdash_{\mathsf{pure}} e : \mathbb{N}}{\Gamma \vdash \mathsf{embed}\ e : \mathbb{N}}$$

Figure 3.18: Syntax and the new typing rule of the λ_{embed}.

With this, we instantiate $\emptyset;\mathbb{N} \models e : \mathbb{N};\mathbb{N}$ (that we get from Lemma 3.4.3) with the empty continuation/metacontinuation, and get the observational refinement between e and $\mathbf{E}[\![e]\!]$. □ □

This completes our treatment of denotational semantics of λ_{delim}. The next section examines interoperability of delimited continuations and other effects.

3.5 Modeling Interoperability Between Languages

A key advantage of using (G)Itrees for semantics is that they can provide a common framework for multi-language interaction. This section presents a case study on the interaction between the languages λ_{embed} (with higher-order store effects) and λ_{delim} (with delimited continuations). Specifically, we allow embedding closed λ_{delim} programs into λ_{embed}, and equip λ_{embed} with a type system that guarantees safe interoperability.

The embedding we provide is restrictive, preventing programs with delimited continuations from accessing outer-language continuations. We leave developing a more permissive type system for future work. At the end of the section we give an example of how to verify a more involved interaction of effects, albeit without the type system.

In this section we reuse the semantics of λ_{delim} from the previous section and higher-order store effects from Section 3.2. For λ_{delim}, we reuse the semantics from the previous section and higher-order store reifiers from [49] In this section, we use magenta to explicitly highlight programs written in λ_{delim}, and for the interpretation functions of the denotational model of λ_{delim}.

Language λ_{embed}. λ_{embed} is a λ-calculus with base types $\mathbb{N}$ and $\mathbf{1}$, references types $\mathsf{ref}(\tau)$ and function types $\tau \to \sigma$, with syntax given in Figure 3.18. Additionally, it includes a construct $\mathsf{embed}\ e$ for embedding λ_{delim} programs. The typing rules are all standard, except for the new typing rule for the embedding.

The idea behind the rule is that we can embed an expression from λ_{delim} if it is a "pure" expression that can evaluate to a natural number. The use of pure typing judgment for the embedded program ensures that it does not to alter the answer type.

$$\boxed{\mathbf{E}[\![-]\!] : \mathit{Expr} \to (\mathit{Var} \to \mathbf{IT}) \to \mathbf{IT}}$$

$$\mathbf{E}[\![\mathsf{embed}\ e]\!]_\rho = \mathsf{Reset}(\mathsf{next}(\mathbf{E}[\![e]\!]\emptyset))$$
$$\mathbf{E}[\![x]\!]_\rho = \rho(x)$$
$$\mathbf{E}[\![\ell]\!]_\rho = \mathsf{Ret}(\ell)$$

$$\mathbf{E}[\![\mathsf{ref}(e)]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e]\!]_\rho, \lambda x.\mathsf{Alloc}(x, \mathsf{Ret}))$$
$$\mathbf{E}[\![!\,e]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e]\!]_\rho, \lambda x.\mathsf{Read}(x))$$
$$\mathbf{E}[\![e_1 \leftarrow e_2]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\ \mathsf{get\_ret}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\mathsf{Write}(y,x)))$$

$$\mathrm{VRel} \triangleq \mathbf{IT}^v \to \mathit{iProp}$$
$$\mathrm{ERel} \triangleq \mathbf{IT} \to \mathit{iProp}$$
$$\mathcal{O} : \mathrm{VRel} \to \mathrm{ERel}$$
$$\mathcal{O}(P)(\beta) \triangleq \mathsf{clwp}\,\beta\,\{x.\ P\,x * \mathsf{has\_state}([])\}$$
$$\mathcal{E} : \mathrm{VRel} \to \mathrm{ERel}$$
$$\mathcal{E}(P)(\beta) \triangleq \mathsf{heap\_ctx} \mathrel{-\!\!*} \mathsf{has\_state}([]) \mathrel{-\!\!*} \mathcal{O}(P)(\beta)$$

$$[\![\tau]\!] : \mathrm{VRel}$$
$$[\![\mathbf{1}]\!](\beta) \triangleq \beta = ()$$
$$[\![\mathbb{N}]\!](\beta) \triangleq \exists n.\, \beta = \mathsf{Ret}(n)$$
$$[\![\tau \to \sigma]\!](\beta) \triangleq \exists F.\, \beta = \mathsf{Fun}(F) \wedge \square\,\forall \beta.\, [\![\tau]\!](\beta) \mathrel{-\!\!*} \mathcal{E}([\![\sigma]\!])(\mathsf{Fun}(F) \bullet \beta)$$
$$[\![\mathsf{ref}(\tau)]\!](\beta) \triangleq \exists \ell.\, \beta = \mathsf{Ret}(\ell) \wedge \boxed{\exists v.\, \ell \mapsto v * [\![\tau]\!](v)}$$

$$[\![\Gamma]\!](\rho) \triangleq \forall (x : \tau \in \Gamma).\, \square\, \mathcal{E}([\![\tau]\!])(\gamma\, x) \qquad \Gamma \vDash e : \tau \triangleq \forall \rho.\, [\![\Gamma]\!](\rho) \mathrel{-\!\!*} \mathcal{E}([\![\tau]\!])(\mathbf{E}[\![e]\!]_\rho)$$

Figure 3.19: Denotational semantics (selected clauses) and logical relation for λ_{embed}.

This means that we can treat an embedded expression as a "complete" program, that does not require outer continuation delimiters, even though it may rely on delimited continuations internally. Those restrictions are crucial for the type safety of the embedding. The typing guarantees that e does not expect any additional delimiters, but it does not, by itself guarantee that any continuations in e escape the embedding boundary. To prevent that we enforce the continuation delimiter along the embedding boundary in the interpretation of embedded expressions.

Denotational model of λ_{embed}. For denotational semantics of λ_{embed}, we start by defining reifiers for the effect signature, which includes higher-order store operations (allocating, reading, and storing references) as $E_{\mathtt{state}}$, and effects related to delimited continuations ($E_{\lambda_{\mathsf{delim}}}$). Then the combined effect signature is $E_{\lambda_{\mathsf{delim}}} \times E_{\mathtt{state}}$, and thus we also let $\mathit{State} \triangleq \mathit{State}_{\mathtt{delim}} \times \mathit{State}_{\mathtt{state}}$, and reifiers are defined component-wise. Figure 3.19 shows the key parts of the denotational semantics. For most of the syntactic constructs we give the standard interpretation. For embed e we use the interpretation $\mathbf{E}[\![-]\!]$ for λ_{delim} from Section 3.4.2, and explicitly wrap the resulting GITree in a Reset. This continuation delimiter acts as a sort of *glue code* to protect the rest of the program from being captured by control operators from the embedded λ_{delim} program.

To show type safety of λ_{embed}, we construct a logical relation (shown in Figure 3.19), which is similar to the other logical relations we considered in this paper, mainly different in the observation relation $\mathcal{O}$. Given that the type system for λ_{embed} effectively prevents expressions of λ_{delim} to access contexts from λ_{embed}, we refine

the observation relation to get access to a version of the WP-HOM rule for expression interpretations of well-typed programs of λ_{embed}, which we do not have in general, as discussed in Section 3.3.

Instead of the standard weakest precondition wp, we utilize a *context-local weakest precondition* clwpre which bakes-in the bind rules [130].

Definition 3.5.1. *Context-local weakest precondition (clwp) is defined as follows:* $\mathit{clwp}\,\alpha\,\{\Phi\} \triangleq \forall \kappa\,(\Psi : \mathbf{IT}^{v} \to \mathit{iProp}).\,(\forall v.\,\Phi\,v \mathrel{-\!\!*} \mathit{wp}\,(\kappa\,v)\,\{\Psi\}) \to \mathit{wp}\,(\kappa\,\alpha)\,\{\Psi\}$

Note that clwp always implies wp and validates a form of the bind rule:

$$\mathsf{clwp}\,\alpha\,\{\beta.\,\mathsf{clwp}\,(\kappa\,\beta)\,\{\Phi\}\} \vdash \mathsf{clwp}\,(\kappa\,\alpha)\,\{\Phi\}$$

for any homomorphism κ. We use the clwp in the definition of observational refinement in the model. Observational refinement asserts that if the evaluation of a top-level expression of λ_{embed} starts with an empty continuation stack, then the evaluation does not introduce new elements into the continuation stack. By using clwp we get a semantical bind lemma, which can be seen as a version of WP-HOM for semantically valid expressions of λ_{embed}. As before, we then obtain fundamental lemma and denotational type soundness.

Lemma 3.5.2. *Semantical bind.* $\mathcal{E}(\lambda x : \mathbf{IT}^{v}.\,\mathcal{E}(P)(\kappa\,x))(\beta)$ *implies* $\mathcal{E}(P)(\kappa\,\beta)$ *for any homomorphism* κ.

Lemma 3.5.3. *Fundamental lemma. Let* $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e : \tau$.

Lemma 3.5.4. *Denotational type soundness. Let* $\emptyset \vdash e : \tau$ *and* $(\mathbf{E}\,[\![e]\!]_{\emptyset}, []) \rightsquigarrow^{*} (\alpha, \sigma)$, *then* $(\exists \beta\ \sigma'.\,(\alpha,\sigma) \rightsquigarrow (\beta,\sigma')) \vee (\alpha \in \mathbf{IT}^{v})$.

Unrestricted interaction of delimited continuations and higher-order state. Even though the type system we considered here is restrictive, we can still reason about unrestricted interactions of events in the "untyped" setting. Here we show an example of such an unrestricted interaction, and demonstrate how to reason about context-dependent and context-independent effects at the same time. While this kind of interactions is forbidden by our type system, we can still write and prove meaningful specifications for such programs. Consider the program in Figure 3.20, written in GITrees directly. The function `prog` utilizes both delimited continuations and state. It takes a reference y as its argument and begins by allocating the value 1 in the store at reference x. Then it captures the continuation $\mathsf{get\_ret}(y, (\lambda l.\mathsf{Let}\,p = \mathsf{NatOp}_{+}(\mathsf{Read}(l), \dots)\,\mathsf{in}\,\mathsf{Write}(l,p)))$ as k. Invoking the continuation k with a number n increments the current value of y by n. The program then invokes this continuation twice. First with the original value of x. Then, with an incremented value of x. Since the starting value of x is 1, the reference y is incremented first by 1 and then by 2. We capture this behavior in the specification for `prog` stated in Figure 3.20.

It is important to note that this program features a bidirectional interaction between state and continuations. Specifically, the body of Shift involves state operations, while

$$
\begin{array}{l}
\texttt{prog} \triangleq \mathsf{Fun}(\mathsf{next}(\lambda y.\\
\quad \mathsf{Let}\, x = \mathsf{Alloc}(\mathsf{Ret}(1))\, \mathsf{in} \qquad \text{Initial offset value}\\
\quad \mathsf{Let}\, n = \mathsf{Shift}(\lambda k.\mathsf{next}(\qquad \text{Capture continuation as } k\\
\qquad \mathsf{Appcont}'(\mathsf{Read}(x), k)\ ; \qquad \text{First call to } k \text{ with the init. value of } x\\
\qquad \mathsf{Let}\, m = \mathsf{NatOp}_+(\mathsf{Read}(x), \mathsf{Ret}(1))\\
\qquad\quad \mathsf{in}\, \mathsf{Write}(x, m)\ ; \qquad x := x+1\\
\qquad \mathsf{Appcont}'(\mathsf{Read}(x), k)))\\
\qquad \mathsf{in}\, \mathsf{get\_ret}(y, (\lambda l.\\
\qquad\quad \mathsf{Let}\, p = \mathsf{NatOp}_+(\mathsf{Read}(l), n)\\
\qquad\quad \mathsf{in}\, \mathsf{Write}(l, p))))) \qquad y := y+n\\
\qquad\quad \text{where } \mathsf{Appcont}'(x,y) \triangleq \mathsf{Appcont}(\mathsf{next}(x), \mathsf{next}(\mathsf{Tau} \circ y \circ \mathsf{next}))
\end{array}
$$

$$
\frac{\mathsf{heap\_ctx} \qquad \mathsf{has\_state}(\sigma) \qquad y \mapsto \mathsf{Ret}(n)}{\mathsf{wp}\,(\mathsf{Reset}(\mathsf{next}(\texttt{prog} \bullet \mathsf{Ret}(y))))\,\{y \mapsto \mathsf{Ret}(n+3) * \mathsf{has\_state}(\sigma)\}}
$$

Figure 3.20: Example program with delimited continuations and state and its specification.

the result of Shift is subsequently used to increment a value in the heap. As we have seen, while this type of interaction is not allowed in the type system, we can still reason about them in program logic. We stipulate that our proposed type system could potentially be extended to support embedding "pure" functions, allowing for bi-directional interaction between the two languages. We believe that such an extension would require implementing answer-type polymorphism, following the approach of Asai and Kameyama [8].

3.6 Discussion and Related Work

We conclude the paper by discussing related work and future directions.

This paper extends GITrees to handle context-dependent effects, which allows us to model higher-order languages with control operators like call/cc (x. e) and $\mathcal{S}$ x. e. We showed this extension supports interoperability between languages with different context-dependent effects, while preserving reasoning about context-independent effects. Our approach leverages the native support for higher-order functions and effects in GITrees. This differs from the first-order effect representation of effects in ITrees [145], which would require explicitly first-order representation of functions and continuations, if we want to model first-class continuation. Such model would mix syntactic and semantic concerns, which is part of what we are trying to avoid by working with (G)ITrees.

Another difference with ITrees is our approach to reasoning. While ITrees use bisimulation-based equational theory, we follow GITrees in using tailored program

logics and defining refinements. Our logic are expressive enough to define logical relations and carry out computational adequacy proofs. In future work, it would be interesting to develop techniques for reasoning about weaker notions of equality than the basic equational theory that GITrees comes equipped with, see the more extensive discussion of this point in [49].

The "classical" domain theory remains an important source of inspiration and ideas for our development, and we want to mention some of the related work along those lines. Cartwright and Felleisen [28] introduced a framework of extensible direct models for constructing modular denotational semantics of programming languages. Their framework centers on an abstract notion of *resources* for representing effects (such as store or continuations) and a central `admin` function that manages these resources. Each language extension defines both the types (domains) of additional values and resources, and specifies the *actions* that the `admin` function can perform on these resources. Building on this framework, Sitaram and Felleisen [117] demonstrated that such models can provide direct-style fully abstract semantics for control operators. Their approach interprets effects, including continuations, by delegating them to a top-level handler. Our work adopts several key ideas from this line of research but reformulates them in the context of GITrees rather than classical domain theory. In our framework, effect signatures define resources, reifiers specify actions, and the reduction relation serves as the central authority dispatching the effects. The transition to GITrees enables us to formalize the extensibility of this approach in a practical manner, and it allows us to develop program logics where "resources" (as above) become resources in separation logic.

Compared to other programming languages paradigms and effects, type systems and logical relations for delimited continuations have not been studied as comprehensively. The original type system for `shift`/`reset` is due to Danvy and Filinski [32], where they employ *answer-type modification*. Materzok and Biernacki [88] generalized this type system to account for more involved control operators $\texttt{shift}_0$/$\texttt{reset}_0$; an alternative substructural type system for these operators was designed by Kiselyov and Shan [77]. Dyvbig, Peyton Jones, and Sabry [40] provide a typed monadic account of CPS for delimited continuations with dynamic prompt generation. Asai and Kameyama [8] present a polymorphic variant of the Danvy and Filinski's type system.

Biernacka and Biernacki [17] prove termination for a language with $\mathcal{S}\ \mathsf{x}.\ e$ and $\mathcal{D}\ \mathsf{e}$ (but without recursion) using logical relations based on abstract machine semantics. The shape of their logical relation is similar to our logical relation used for showing adequacy in that they also have relations for configurations, metacontinuations, etc. Asai [7] uses type-directed logical relations to verify a direct-style specializer (partial evaluator) for a language with $\mathcal{S}\ \mathsf{x}.\ e$ and $\mathcal{D}\ \mathsf{e}$, proving correctness against evaluation-context based operational semantics. In contrast with those works, we define our logical relations on denotations, using the semantics of GITrees and the derived program logics.

In our interoperability example we showed type safety of a combined language, with respect to denotational semantics. In future work we would like to examine other properties: for example, that λ_{delim} expressions cannot disrupt λ_{embed}'s control flow,

perhaps establishing some form of well-bracketedness as in [132].

We would also like to study other context-dependent effects like exceptions, handlers, and algebraic effects [12, 14, 101, 135, 143]. In particular, it would be interesting to give a denotational semantics to a language with handlers, derive program logic rules for it, and compare the resulting program logic to the one in [34].

CHAPTER 4

Solving Guarded Domain Equations in Presheaves Over Ordinals and Mechanizing It

Abstract

Constructing solutions to recursive domain equations is a well-known, important problem in the study of programs and programming languages. Mathematically speaking, the problem is finding a fixed point (up to isomorphism) of a suitable functor over a suitable category. A particularly useful instance, inspired by the step-indexing technique, is where the functor is over (a subcategory of) the category of presheaves over the ordinal ω and the functors are locally-contractive, also known as *guarded functors*. This corresponds to step-indexing over natural numbers. However, for certain problems, *e.g.*, when dealing with infinite non-determinism, one needs to employ trans-finite step-indexing, *i.e.*, consider presheaf categories over higher ordinals. Prior work on trans-finite step-indexing either only considers a very narrow class of functors over a particularly restricted subcategory of presheaves over higher ordinals, or treats the problem very generally working with sheaves over an arbitrary complete Heyting algebra with a well-founded basis.

In this paper we present a solution to the guarded domain equations problem over *all* guarded functors over the category of *presheaves* over ordinal numbers, as well as its mechanization in the Rocq Prover. As the categories of sheaves and presheaves over ordinals are equivalent, our main contribution is simplifying prior work from the setting of the category of sheaves to the setting of the category of presheaves and mechanizing it — presheaves are more amenable to mechanization in a proof assistant.

4.1 Introduction

Recursive Domain Equations and Step-Indexing Recursive definitions are prevalent in computer programming. Thus, one of the important problems in the

study of programs and programming languages is finding recursive mathematical objects to construct models of programs or the mathematical tools to study them, *e.g.*, program logics. This problem is often stated as a so-called *domain equation* [118] in terms of a fixed point (up to isomorphism) of an endo-functor $F : \mathcal{C} \to \mathcal{C}$ on a suitable category $\mathcal{C}$, *i.e.*, an object X of $\mathcal{C}$ such that $F(X) \simeq X$.[1] The problem was first studied by Dana Scott [107, 108] in the category of continuous lattices in order to give denotational semantics to untyped λ-calculi. Scott's construction [108] takes the inverse limit of an ω-tower of continuous lattices obtained through successive applications of the functor. As observed by Lawvere [108], the essence of the proof showing that this construction indeed constructs a fixed point is that the inverse limit coincides with the direct limit (of a related diagram). This has since been named the limit-co-limit coincidence theorem [118]. Wand [139] later observed that the essential point, rather than the category itself, is the structure of its hom-sets. Wand [139], Smyth, and Plotkin [118] give an abstract account of solving domain equations for endo-**O**-functors on **O**-categories, *i.e.*, categories that are enriched over the category of ω-cpos and ω-continuous functions and functors over them whose actions on morphisms is ω-continuous.[2] America and Rutten [5] solve domain equations in a certain category of metric spaces. Birkedal *et al.* [22] generalize the results of America and Rutten by constructing solutions to domain equations over so-called **M**-categories, categories enriched over the category of ultra-metric spaces and non-expansive maps, where the functors considered are locally contractive in the sense that the functors' action on morphisms is a contractive function. This generalization is inspired [22] by the relationship between bounded bisected ultra-metric spaces (where the distances belong to the set $\{0\} \cup \left\{ \frac{1}{2^n} \middle| n \in \mathbb{N} \right\}$) and the technique of step-indexing [2, 6, 23, 90]. Birkedal *et al.* [24] later generalize these results further to a setting where the category is enriched over the category of sheaves over a complete Heyting algebra with a well-founded basis; ordinals in general, and ω in particular, being such complete Heyting algebras. These results [22, 24] have served as a foundation for a multitude of works based on step-indexing, *e.g.*, to give denotational semantics to programs [30, 93], to construct the model of the Iris program logic framework [73].

The Need for Step-Indexing Over Higher Ordinals It is well known [25, 27] that if one uses the step-indexing technique to reason about a programming language with countable non-determinism, it is no longer sufficient to consider step-indexing over ω. One must [27] instead use step-indexing over ω_1 (the first uncountable ordinal). Another way to look at this issue is through the lens of the step-indexed logic. The pertinent problem to consider here is the question of when existence of an object inside the step-indexed logic implies its existence outside. This is dubbed "the existential property" by Spies *et al.* [119]. Say we are given a predicate ϕ over a set A for which we have that $\exists x : A.\ \phi(x)$ is a valid sentence in the model of the step-indexed

[1] In this paper we assume the reader is familiar with basic concepts in category theory found in standard textbooks [11, 83].

[2] See Section 4.3 for a brief explanation of enriched categories and functors. More details can be found in the book of Kelly [74] on the subject.

logic, *i.e.*, $\models \exists x : A.\ \phi(x)$. The existential property states that $\models \exists x : A.\ \phi(x)$ implies that there exists some $a \in A$ for which we have $\models \phi(a)$. Question: when does the existential property hold? Answer: if the cardinality $|A|$ is strictly smaller than that of the ordinal γ we are step-indexing over (provided that γ is a regular ordinal). Indeed, if $|A|$ is not smaller than the step-indexing ordinal γ, there are predicates for which the existential property does not hold. For a detailed, formal discussion of the need for step-indexing over higher-ordinals see Appendix A.1.

Spies *et al.* [119], motivated by this need, extend the step-indexed logic underlying the Iris program logic to trans-finite ordinals. Their work is also mechanized on top of the Rocq Prover. However, their domain equation solver can only be used to solve domain equations of functors of a special form. This special form is suitable for constructing Iris's higher-order resources [72], but is not sufficient for arbitrary locally contractive functors, *e.g.*, the functor for constructing so-called guarded interaction trees [49]. We will further discuss the relation between our work and Spies *et al.* [119] in Section 4.6.

Step-Indexing Over All Ordinals In the rest of this paper we talk about step-indexing over (all) ordinals, *e.g.*, we speak of sheaves or presheaves over **Ord**, the set of all ordinals (which we also consider to be a preorder category under the usual order). This is to be understood as the set of all ordinals definable in a certain Grothendieck universe. (In our Rocq formalization, the type **Ord** is a universe polymorphic definition corresponding to the type of all ordinals in the universe.) In terms of (pre)-sheaves over **Ord**, this should be understood as (pre)-sheaves over the ordinal that is the supremum of all ordinals in **Ord** — which obviously itself lives in a version of **Ord** in a larger Grothendieck universe. The upshot of step-indexing over all ordinals in the universe is that then the existential property holds for any set/type in the universe (see Appendix A.1).

The only downside of working with **Ord** is that it is not closed under suprema. That is, there are subsets $A \subseteq \mathbf{Ord}$ such that $\sup(A)$ does not exist (technically it does but it lives in a copy of **Ord** in a larger universe). To compensate for this issue, many of our definitions and constructions are parameterized by an arbitrary *downwards-closed* subset of ordinals instead of ordinals. This can intuitively be thought of as working with *the completion of* **Ord** as a lattice instead of **Ord** itself. This significantly simplifies the presentation, and more importantly mechanization of our results; see Section 4.5.

Equivalence of Categories of Sheaves Over Ordinals and Presheaves Over Ordinals As we will discuss in Section 4.2 the category of sheaves over ordinals, $\mathbf{Sh}(\mathbf{Ord})$, is equivalent (in fact adjoint equivalent) to the category of presheaves over **Ord**, $\mathbf{PSh}(\mathbf{Ord})$. Thus, technically speaking, the results of Birkedal *et al.* [24] subsume the results we present in this paper. That is, since Birkedal *et al.* [24] construct solutions to guarded domain equations over sheaves over complete Heyting algebras with a well-founded basis, and ordinals are a particular instance of such

Heyting algebras, one can obtain solutions to equations over **PSh**(**Ord**) from solutions to equations over **Sh**(**Ord**). This is similar to how one obtains solutions to classical domain equations (over category **Dom** of domains) from those over the equivalent category **CUSL** of conditional upper semi-lattices with a least elements [125, chapter 4].

Contributions Notwithstanding the point above regarding the equivalence of categories of presheaves and sheaves over ordinals, the main contribution of this paper is simplifying and mechanizing the results of Birkedal *et al*. [24] to the setting of presheaves over ordinals and locally contractive functors on them which is much more amenable to mechanization. In fact, the aforementioned equivalence is the reason we were convinced that a simplification to the setting of presheaves, and a direct solution construction in that setting is achievable and mechanizable. In this paper we present this simplified version and its mechanization in the Rocq Prover. We also mechanize the symmetrization argument [48] to solve mixed-variance recursive domain equations and provide an example of solving a concrete mixed-variance equation using our framework. All results marked with  are mechanized [121] in the Rocq Prover. We only present a few high-level proofs in this paper which help the reader appreciate the results. For the rest we refer to our Rocq mechanization [121].

The Structure of the Rest of the Paper Section 4.2 introduces some basic constructions over the category of sheaves and presheaves over ordinals, including their equivalence. In Section 4.3 we present categories enriched over (the cartesian structure of) presheaves over ordinals, including the central concepts of enriched and locally contractive functors. We also present the concepts of ordinal-partial isomorphism and enriched-pointwiseness of limits, which play an important role in our construction of solutions of domain equations. Section 4.4 gives details of our construction of solutions to domain equations as well as their uniqueness. The technicalities involved in the mechanization of the results are discussed in Section 4.5. In Section 4.6 we discuss other works related to ours, and present our future work and concluding remarks in Sections 4.7 and 4.8 respectively. The Appendix A.1 presents discusses the need for higher ordinals, while Appendices A.2 to A.4 include some details omitted from the main text.

Remark 4.1.1 (Notation)**.** *We fix the following notational convention for the rest of the paper:*

Notation	*Convention / Meaning*
$X := Y$	*X is defined as Y*
$\alpha, \beta, \gamma, \ldots$	*Ordinals*
α^+	*Successor of* α
$\alpha \prec \beta, \alpha \preceq \beta$	*Order of ordinals*
$F, G, H, \ldots$	*(Pre)sheaves, functors*
$A, B, C, \ldots$	*Objects, could be (pre)sheaves*
$\eta, \xi, \zeta, \ldots$	*Natural transformations*
$f, g, h, \ldots$	*Morphisms, could be natural transformations*
Π_A^L	*Projection from L onto* $F(A)$ *when L is the limit of the functor F*
$F\|_A$	*Restrict the domain of the functor (or function) F to A*
$\mathcal{Y}$	*The Yoneda embedding*
$A \simeq B$	*Isomorphism*
$f : A \xrightarrow{\simeq} B$	*The morphism* $f : A \to B$ *has an inverse and A and B are isomorphic*
$F_{\beta \preceq \alpha}$	*The map* $F(\beta) \to F(\alpha)$ *induced by the (pre)sheaf F*
$\lim_{\beta \prec \alpha} F(\beta)$	*Limit of the diagram F whose domain is (restricted to)* $\{\beta \mid \beta \prec \alpha\}$*; when obvious, we drop the* $\beta \prec \alpha$ *part*
$\lim_{\beta} f_\beta$	*The unique morphism from A to* $\lim_{\beta} F(\beta)$ *when* $f_\beta : A \to F(\beta)$
$curry(f)$	*The exponential transpose of f*

4.2 On Sheaves and Presheaves Over Ordinals

We start by giving a few basic definitions in the category of presheaves over ordinals, $\mathbf{PSh}(\mathbf{Ord})$. In particular, there are two important endo-functors on $\mathbf{PSh}(\mathbf{Ord})$ called *later* ($\blacktriangleright : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$), and *earlier* ($\blacktriangleleft : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$). These functors are defined as follows:

$$\blacktriangleright F(\alpha) := \lim_{\beta \prec \alpha} F(\beta) \qquad \blacktriangleleft F(\alpha) := F(\alpha^+)$$
$$(\blacktriangleright F)_{\beta \preceq \alpha} := \lim_{\gamma \prec \beta} \Pi_\gamma^{\blacktriangleright F(\alpha)} \qquad (\blacktriangleleft F)_{\beta \preceq \alpha} := F_{\beta^+ \preceq \alpha^+}$$

The object map of $\blacktriangleright$, at each stage, takes the limit (in **Set**) of the diagram induced by the object (presheaf) it is mapping at all smaller stages. In particular, $\blacktriangleright F(0)$ is always the terminal (singleton) set, and $\blacktriangleright F(\alpha^+) \simeq F(\alpha)$ (see Lemma A.4.1 in Appendix A.4). The morphism map of the functor $\blacktriangleright$, $(\blacktriangleright F)_{\beta \preceq \alpha}$ is defined as the

amalgamation of projections $\Pi_\gamma^{\blacktriangleright F(\alpha)} : \blacktriangleright F(\alpha) \to F(\gamma)$ of the limit that is $(\blacktriangleright F)(\alpha)$. The functoriality of $\blacktriangleleft$ is trivial. The functoriality of $\blacktriangleright$, on the other hand, follows from properties of limits. It is well-known that these two functors, later and earlier, form an adjunction [24]: $\blacktriangleleft \dashv \blacktriangleright$.

There is an important natural transformation $\mathsf{Next} : \mathsf{id}_{\mathbf{PSh}(\mathbf{Ord})} \to \blacktriangleright$. The map (morphism in **Set**) $\mathsf{Next}_F(\alpha) : F(\alpha) \to \blacktriangleright F(\alpha)$ is constructed as follows: given an element $x \in F(\alpha)$, $(\{*\}, \{F_{\beta \prec \alpha}\}_{\beta \prec \alpha})$ is a cone on the diagram $F|_{\{\beta | \beta \prec \alpha\}}$ in **Set** — the vertex of this cone, $\{*\}$, is the terminal object of **Set**. Since $\blacktriangleright F(\alpha)$ is the limit of this diagram, there is a unique map from the $\{*\}$ into $\blacktriangleright F(\alpha)$. We take the image this map to be the result of Next:

$$\mathsf{Next}_F(\alpha)(x) := \left(\lim_{\beta \prec \alpha} N_\beta^x\right)(*) \text{where} N_\beta^x(*) := F_{\beta \prec \alpha}(x)$$

That this construction is natural both in F and α, like most properties that are related to later, naturality follows from properties of limits.

4.2.1 Equivalence of the Category of Sheaves Over Ordinals and the Category of Presheaves Over Ordinals

Sheaves are presheaves that additionally satisfy the so-called "sheaf condition" [87]. In the particular case of ordinals (seen as a topological space) the sheaf condition boils down to the following: at any limit ordinal, including zero, the value of the sheaf must be the (categorical) limit of all the sets below it. That is, a presheaf $F : \mathbf{Ord}^{\mathsf{op}} \to \mathbf{Set}$ is a sheaf if and only if we have both that $F(0) \simeq \{*\}$ and that $F(\lambda) \simeq \lim_{\alpha \prec \lambda} F(\alpha)$ via mediating morphisms, for any limit ordinal λ.

As per the sheaf condition above, by construction, $\blacktriangleright F$ is always a sheaf, regardless of F. Thus, $\blacktriangleright$ is also a functor from the category of presheaves over ordinals the category of sheaves over ordinals. On the other hand, $\blacktriangleleft F$ need not be a sheaf, even if F is. When viewed as functors between the category of sheaves and presheaves, the earlier and later functors form not only an adjunction, as noted above, but an adjoint equivalence.[3] That is, the following isomorphisms hold and are both natural in F:

$$\blacktriangleleft(\blacktriangleright(F)) \simeq F \quad \text{for any } \textit{presheaf } F \quad \text{and} \quad \blacktriangleright(\blacktriangleleft(F)) \simeq F \quad \text{for any } \textit{sheaf } F$$

These isomorphisms and their naturality rely on Lemma A.4.1 in Appendix A.4.

Remark 4.2.1. *The discussion above of the adjoint equivalence of* $\mathbf{Sh}(\mathbf{Ord})$ *and* $\mathbf{PSh}(\mathbf{Ord})$ *in fact holds generally for any limit ordinal* λ*,* i.e.*, for showing adjoint equivalence of* $\mathbf{Sh}(\lambda)$ *and* $\mathbf{PSh}(\lambda)$.

4.2.2 Contractive Morphisms and Their Fixed Points

Here, we define contractive morphisms in the category of presheaves (natural transformations) and show that they always have unique fixed points — the construction and

[3] This equivalence was noticed in a discussion the second author had with Daniel Gratzer.

the proof are very similar to the classical Banach fixed point theorem. Fixed points of contractive morphism are useful in defining so-called guarded recursive predicates which are particularly useful when working in step-indexed logics [21]. In addition to this, we present contractive morphisms and construction of their fixed points not only to highlight the difference in the construction compared to Birkedal *et al.* [24], but also because they are used in proving uniqueness of solutions of domain equations — an important fact in our development; see Section 4.4.3.

We say a morphism in **PSh**(**Ord**), *i.e.*, a natural transformation, is contractive, if it factors through Next. We write $\mathsf{ContrMorph}(\eta)$ when η is contractive.

Definition 4.2.2 ( ). *A natural transformation $\eta : F \to G$ is contractive,* i.e., $\mathsf{ContrMorph}(\eta)$, *if there is a natural transformation $\eta' : \blacktriangleright F \to G$ such that $\eta = \eta' \circ \mathsf{Next}_F$. We call η' a* witness *of contractivity of η.*

Lemma 4.2.3 ( ). *Let $\eta : F \to G$ be a contractive morphism,* i.e., $\mathsf{ContrMorph}(\eta)$, *with η' a witness of contractivity. Then the following holds for any $\xi : H \to F$ and any $\zeta : G \to H$:*

$\mathsf{ContrMorph}(\eta \circ \xi)$	*witnessed by* $\eta' \circ \blacktriangleright \xi$
$\mathsf{ContrMorph}(\zeta \circ \eta)$	*witnessed by* $\zeta \circ \eta'$

Definition 4.2.4 (Fixed Points). *We define three notions of fixed points of morphisms:*

1. *$\xi : B \to A$ is a fixed point of $\eta : \blacktriangleright A \times B \to A$ if $\eta \circ \langle \mathsf{Next}_A \circ \xi, \mathsf{id}_B \rangle = \xi$*
2. *$\xi : 1 \to A$ is a fixed point of $\eta : \blacktriangleright A \to A$ if $\eta \circ \mathsf{Next}_A \circ \xi = \xi$*
3. *$\xi : 1 \to A$ is a fixed point of the contractive morphism $\eta : A \to A$ if $\eta \circ \xi = \xi$*

Remark 4.2.5 ( ). *Definition 4.2.4 defines three kinds of fixed points each weaker than the one before in that if (unique) solutions to one kind of fixed point exist, so do (unique) solutions to the next kind. Theorem 4.2.6 below immediately implies existence of unique fixed points of the first kind and thus existence of unique fixed points of all kinds. We will write* $\mathsf{fix}(f)$ *for the* unique *fixed point of map f for any of these kinds of fixed points.*

In order to construct these fixed points we show that there is a general fixed point combinator $fix_A : A^{\blacktriangleright A} \to A$. Note that here the fixed point combinator fix_A is a natural transformation (a morphism in the category of presheaves) from the exponential object $A^{\blacktriangleright A}$ to A.

Theorem 4.2.6 ( ). *For any presheaf A, there is a natural transformation $fix_A : A^{\blacktriangleright A} \to A$ in the category of presheaves over ordinals that acts as the fixed point combinator constructing unique fixed points. That is, for any $\eta : \blacktriangleright A \times B \to A$ we have $fix_A \circ curry(\eta)$ is the unique natural transformation from B to A such that: $\eta \circ \langle \mathsf{Next}_A \circ fix_A \circ curry(\eta), \mathsf{id}_B \rangle = fix_A \circ curry(\eta)$ where $curry(\eta)$ is the exponential transpose of η.*

Remark 4.2.7. *The proof of Theorem 4.2.6 differs from the proof given by Birkedal* et al. *[24] in that working in the category of sheaves, the value of the fixed point presheaf is uniquely determined at 0 and all limit ordinals. In contrast, our construction applies η at every single stage of the construction including at 0 and limit ordinals. In other words, at 0 and limit ordinals, we apply η* "one more time" *after computing what one would compute in the case of sheaves. To see this, note that a natural transformation $fix_A : A^{\blacktriangleright A} \to A$ essentially amounts to maps (morphisms in* **Set***) $\zeta_\alpha : (\mathcal{Y}_\alpha \times \blacktriangleright A \to A) \to A(\alpha)$ that are natural in α — each ζ_α is a map from the set of natural transformations $(\mathcal{Y}_\alpha \times \blacktriangleright A \to A)$ to the set $A(\alpha)$. Thus, at 0, we are given a function, say $f : \mathcal{Y}_\alpha(0) \times \blacktriangleright A(0) \to A(0)$ and need to produce an element of $A(0)$, for which we will use f. Intuitively, the function f here is the natural transformation η at stage 0,* i.e.*, the natural transformation we are taking the fixed point of.*

4.3 Enrichment Over Categories of Presheaves Over Ordinals

Enrichment is often studied over monoidal categories [74]. Here, we work specifically with the monoidal structure of the cartesian closedness of the enriching category, *i.e.*, the category **PSh**(**Ord**). We briefly present the basic definitions here just to fix notation. Our notion of locally contractive functor is exactly that in Birkedal *et al.* [24].

4.3.1 Enriched Categories and Functors; Locally Contractive Functors

Definition 4.3.1 (Enriched Category). *We say a category $\mathcal{C}$ is enriched over a cartesian closed category $\mathcal{E}$ if we have the following:*

- *An internal hom object in $\mathcal{E}$, written $\mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{A,B}$, for any pair of objects A and B in $\mathcal{C}$*
- *A map $\lceil\cdot\rceil : Hom_\mathcal{C}(A,B) \to Hom_\mathcal{E}(1, \mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{A,B})$ embedding $\mathcal{C}$ morphisms into $\mathcal{E}$*
- *A map $\lfloor\cdot\rfloor : Hom_\mathcal{E}(1, \mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{A,B}) \to Hom_\mathcal{C}(A,B)$ projecting $\mathcal{C}$ morphisms out of $\mathcal{E}$*
- *The maps $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ are inverses of one another*
- *Internal composition morphisms: $\mathbb{E}^{\mathsf{comp}_\mathcal{C}}_{A,B,C} : \mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{A,B} \times \mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{B,C} \to \mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{A,C}$*
- *Expressed in terms of equality of morphisms in $\mathcal{E}$, we have that $\mathbb{E}^{\mathsf{comp}_\mathcal{C}}_{A,B,C}$ is in agreement with composition in $\mathcal{C}$, is associative and respects identity morphisms*

Definition 4.3.2 (Enriched Functor). *Let $\mathcal{C}$ and $\mathcal{D}$ be two $\mathcal{E}$-enriched categories. We say a functor $F : \mathcal{C} \to \mathcal{D}$ is $\mathcal{E}$-enriched if there are morphisms $\mathbb{E}^{\mathsf{hm}_F}_{A,B} : \mathbb{E}^{\mathsf{hom}_\mathcal{C}}_{A,B} \to \mathbb{E}^{\mathsf{hom}_\mathcal{D}}_{F(A),F(B)}$ in $\mathcal{E}$ that acts as the $\mathcal{E}$-internal functor action of F and, expressed in terms of equality of morphisms in $\mathcal{E}$, the morphisms $\mathbb{E}^{\mathsf{hm}_F}_{A,B}$ preserve identity and composition.*

Following Birkedal *et al*. [24] we define a locally contractive functor to be an enriched functor (over the category of presheaves) that also has a *contracted* internal functor action morphism. Intuitively, what we want is to say that a functor is locally contractive if its internal functor action is a contractive morphism in the sense of Definition 4.2.2. The definition below furthermore requires the witness of contractivity of the internal functor action to also act functorially in the sense that it must also preserve compositions and identities. This extra requirement, as also pointed out by Birkedal *et al*. [24], is the reason why we can develop the theory of ordinal-partial isomorphisms and enriched-pointwise limits as we present in this section. Birkedal *et al*. [24] present the theory of ordinal-partial isomorphisms but do not make enriched-pointwise limits formal — they only mention intuitively that "since limits are computed pointwise, ..." when presenting their approach to constructing solutions to domain equations. Because we mechanize our solution to the domain equation problem we had to formalize and mechanize this intuitive line of argument.

Definition 4.3.3 (Locally Contractive Functors). *Let $\mathcal{C}$ and $\mathcal{D}$ be two* $\mathbf{PSh}(\mathbf{Ord})$*-enriched categories. We say a* $\mathbf{PSh}(\mathbf{Ord})$*-enriched functor $F : \mathcal{C} \to \mathcal{D}$ is locally contractive if the internal action morphisms of F, $\mathbb{E}^{\mathsf{hm}_F}_{A,B}$, are contractive with the witness of contractivity being morphisms $\mathbb{E}^{\blacktriangleright \mathsf{hm}_F}_{A,B} : \blacktriangleright \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{A,B} \to \mathbb{E}^{\mathsf{hom}_{\mathcal{D}}}_{F(A),F(B)}$. Furthermore, expressed in terms of equality of morphisms in* $\mathbf{PSh}(\mathbf{Ord})$*, the morphisms $\mathbb{E}^{\blacktriangleright \mathsf{hm}_F}_{A,B}$ must preserve identity and composition.*

Lemma 4.3.4 (Composition of Enriched and Locally Contractive Functors). *Let $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{B}$ be three $\mathcal{E}$-enriched categories and $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{B}$ be two $\mathcal{E}$-enriched functors. The composition $G \circ F$ is also an $\mathcal{E}$-enriched functor. Furthermore, $G \circ F$ is locally contractive if at least one of F or G is.*

Enriched functors are closed under many useful constructions: constant functors, the identity functor, products of functors, their sums, diagonal functors ($\Delta_n : A \mapsto A^n$), *etc*. In particular, this includes all finitary polynomial functors. Lemma 4.3.4 shows that there also exists a similarly large collection of locally contractive functors because the later functor, $\blacktriangleright : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$ is both enriched and locally contractive; see Appendix A.2 where we also show that the earlier functor $\blacktriangleleft : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$ is not even enriched, let alone locally contractive.

4.3.2 Ordinal-Partial Isomorphisms

In this section, following Birkedal *et al*. [24], we define a notion of ordinal-partial isomorphisms, indexed over ordinals, for categories enriched over $\mathbf{PSh}(\mathbf{Ord})$ and prove a few useful lemmas about such morphisms that we will later use in solving domain equations.

Definition 4.3.5 (Ordinal-Partial Isomorphism). *Let $\mathcal{C}$ be a category enriched over* $\mathbf{PSh}(\mathbf{Ord})$ *and $f : A \to B$ be a morphism in $\mathcal{C}$. We say that f is an α-isomorphism if*

we have an element $x \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{B,A}(\alpha)$ *called partial inverse of* f *at stage* α *such that*

$$\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{B,A,B}(\alpha)(x, \lceil f \rceil(\alpha)(*)) = \lceil \mathsf{id}_B \rceil(\alpha)(*) \qquad \text{(part-iso-left-id)}$$
$$\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{A,B,A}(\alpha)(\lceil f \rceil(\alpha)(*), x) = \lceil \mathsf{id}_A \rceil(\alpha)(*) \qquad \text{(part-iso-right-id)}$$

By functoriality of $\mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{B,A}$ and naturality of $\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{A,B,A}$ and $\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{B,A,B}$, we know that if f is an α-isomorphism it is also β-isomorphism for any $\beta \preceq \alpha$. Given a downwards-closed subset of ordinals $A \subseteq \mathbf{Ord}$, we say a morphism f is an A-isomorphism if it is an α-isomorphism for any $\alpha \in A$. Whenever A has a maximal element γ being an A-isomorphism is equivalent to being a γ-isomorphism. However, A-isomorphisms are in general useful for working with morphisms that are A-isomorphisms for an unbounded downwards-closed subset A. Intuitively, f being an α-isomorphism means that it behaves like an isomorphism up to stage α, even though an inverse morphism may not even exist.

Remark 4.3.6. *Although we define ordinal-partial isomorphisms almost exactly as Birkedal* et al. *[24] do, due to the differences between sheaves and presheaves, in the setting of Birkedal* et al. *[24] every morphism is a 0-isomorphism, and also any morphism that is a* $\{\alpha \mid \alpha \prec \lambda\}$*-isomorphism for some limit ordinal* λ *is also a* λ*-isomorphism. This is not the case in our setting.*

Lemma 4.3.7 (🚒)**.** *Let* $\mathcal{C}$ *be a* $\mathbf{PSh}(\mathbf{Ord})$*-enriched category and let* $f : A \to B$ *a morphism in* $\mathcal{C}$*. The morphism* f *is an isomorphism,* i.e.*, there is a morphism* $g : B \to A$ *such that* $f \circ g = \mathsf{id}_B$ *and* $g \circ f = \mathsf{id}_A$*,* if and only if *$f$ is an α-isomorphism for all* $\alpha \in \mathbf{Ord}$*.*

Lemma 4.3.8 (🚒)**.** *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be two* $\mathbf{PSh}(\mathbf{Ord})$*-enriched categories and* $F : \mathcal{C} \to \mathcal{D}$ *an* $\mathbf{PSh}(\mathbf{Ord})$*-enriched functor. For any* α*-isomorphism* $f : A \to B$ *in* $\mathcal{C}$*,* $F(f)$ *is an* α*-isomorphism.*

Lemma 4.3.9 (🚒)**.** *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be two* $\mathbf{PSh}(\mathbf{Ord})$*-enriched categories and* $F : \mathcal{C} \to \mathcal{D}$ *a locally contractive functor. Furthermore, let* $f : A \to B$ *in* $\mathcal{C}$ *be a* $\{\beta \mid \beta \prec \alpha\}$*-isomorphism. The image of* f *under* F*,* $F(f)$*, is an* α*-isomorphism in* $\mathcal{D}$*.*

4.3.3 Enriched-Pointwise Limits

In this section we develop the theory of enriched-pointwise limits in categories enriched over $\mathbf{PSh}(\mathbf{Ord})$. This is an abstract way of representing the idea that limits are suitably "pointwise". In particular, when we consider the self-enrichment of $\mathbf{PSh}(\mathbf{Ord})$ and the enrichment of $(\mathbf{PSh}(\mathbf{Ord}))^{\mathsf{op}}$ over $\mathbf{PSh}(\mathbf{Ord})$ this notion directly corresponds to (co-)limits in $\mathbf{PSh}(\mathbf{Ord})$ being pointwise; see Appendix A.4.3. We will use enriched-pointwise limits to state and prove the important Lemma 4.3.13 and Corollary 4.3.14.

Definition 4.3.10 (Enriched-Pointwise Cones 🚒)**.** *Let* $\mathcal{C}$ *be a category enriched over* $\mathbf{PSh}(\mathbf{Ord})$ *and* $F : \mathcal{J} \to \mathcal{C}$ *be a* $\mathcal{J}$*-shaped diagram in* $\mathcal{C}$*. Given an ordinal* α*, an*

enriched-pointwise cone $(V, \{x_j\}_{j\in J})$ *at stage* α *over the diagram* F *consists of a vertex object* V *in* $\mathcal{C}$ *together with elements* $x_j \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{V,F(j)}(\alpha)$ *such that for any morphism* $f : j \to j'$ *in* $\mathcal{J}$ *we have* $\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{V,F(j),F(j')}(\alpha)(x_j, \lceil F(f)\rceil(\alpha)(*)) = x_{j'}$.

Since $F(j)$ is a functor (presheaf) and $\mathbb{E}^{\mathsf{comp}_F}_{V,F(j),F(j')}$ is natural, a cone at stage α is also a cone at stage $\beta \preceq \alpha$. In addition, given a cone $(V, \{S_j\}_{j\in\mathcal{J}})$ over a diagram $F : \mathcal{J} \to \mathcal{C}$ we obtain an enriched-pointwise cone $(V, \{\lceil S_j\rceil(\alpha)\}_{j\in\mathcal{J}})$ of diagram F at any stage α.

Definition 4.3.11 (Enriched-Pointwise Cone Homomorphisms). *Let* $\mathcal{C}$ *be a category enriched over* **PSh**(**Ord**) *and* $F : \mathcal{J} \to \mathcal{C}$ *be a* $\mathcal{J}$*-shaped diagram in* $\mathcal{C}$*. Moreover, let* $(V, \{x_j\}_{j\in J})$ *and* $(V', \{x'_j\}_{j\in J})$ *be two enriched-pointwise cones over* F *both at stage* α*. A cone homomorphism from* V *to* V' *is an element* $h \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{V,V'}(\alpha)$ *such that* $\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{V,V',F(j)}(\alpha)(h, x'_j) = x_j$.

Definition 4.3.12 (Enriched-Pointwise Limits). *Let* $\mathcal{C}$ *be a category enriched over* **PSh**(**Ord**) *and* $F : \mathcal{J} \to \mathcal{C}$ *be a* $\mathcal{J}$*-shaped diagram in* $\mathcal{C}$*. Furthermore, let* $(V, \{S_j\}_{j\in\mathcal{J}})$ *be a limit cone of diagram* F *in* $\mathcal{C}$*. We say this limit is enriched-pointwise if we have that for any enriched-pointwise cone* $(W, \{x_j\}_{j\in\mathcal{J}})$ *at stage* α *there is a unique enriched-pointwise cone homomorphism from* $(W, \{x_j\}_{j\in\mathcal{J}})$ *to the enriched-pointwise cone* $(V, \{\lceil S_j\rceil(\alpha)\}_{j\in\mathcal{J}})$.

Lemma 4.3.13 (). *Let* $\mathcal{J}$ *be a strongly connected preorder category,* i.e.*, for any two objects* $j, j' \in \mathcal{J}$*,* $Hom_{\mathcal{J}}(j, j') \cup Hom_{\mathcal{J}}(j', j) \neq \emptyset$*. Furthermore, let* $\mathcal{C}$ *be a category enriched over* **PSh**(**Ord**) *and* $F : \mathcal{J} \to \mathcal{C}$ *be a diagram such that for any* $f : j \to j'$ *in* $\mathcal{J}$*, the morphism* $F(f)$ *is an* α*-isomorphism. Finally, let the limit of* F *be enriched-pointwise in the sense of Definition 4.3.12. Under these circumstances every projection of the limit of* F *is an* α*-isomorphism.*

Corollary 4.3.14 (). *Let* $\mathcal{C}$ *a category enriched over* **PSh**(**Ord**) *and* $F : \{\beta \mid \beta \prec \alpha\} \to \mathcal{C}$ *be a diagram whose limit is enriched-pointwise. Fix* $\delta \prec \alpha$*. Assume that for any* $\delta \preceq \gamma \prec \alpha$ *the morphism* $F_{\delta\preceq\gamma}$ *is a* δ*-isomorphism. The projection* $\Pi_\delta : \lim F \to F(\delta)$ *is a* δ*-isomorphism.*

4.4 Solving Domain Equations

In this section we show that for a **PSh**(**Ord**)-enriched category $\mathcal{C}$, any locally contractive functor $F : \mathcal{C} \to \mathcal{C}$ has a unique solution up to isomorphism.

4.4.1 Uniqueness of Solutions up to Isomorphism

By a well-known result attributed to Lambek [118], the initial F-algebra for a functor F is an isomorphism, and hence a solution to the domain equation for F. The following

theorem establishes the converse for locally contractive functors showing uniqueness of solutions.

Theorem 4.4.1 (). *Let $\mathcal{C}$ be a* **PSh**(**Ord**)*-enriched category and $F : \mathcal{C} \to \mathcal{C}$ a locally contractive functor. The F-algebra (S, s) induced by a solution $s : F(S) \xrightarrow{\simeq} S$ is an initial algebra.*

Proof. The proof we have mechanized is the exact proof given by Birkedal *et al.* [22, 24]. Given an F-algebra (A, ϕ_A) we need to construct a unique F-algebra morphism from (S, s) to (A, ϕ_A). Observe that a morphism $h : S \to A$ is an F-algebra morphism if and only if we have $h = \phi_A \circ F(h) \circ s^{-1}$. A different way to look at this fact is that given a morphism $h : S \to A$, we can construct another morphism from S to A by taking $\phi_A \circ F(h) \circ s^{-1}$. This mapping induces the following morphism in **PSh**(**Ord**):

$$\mu := compL_A^{\phi_A} \circ compR_S^{s^{-1}} \circ \mathbb{E}_{S,A}^{\mathsf{hm}_F} : \mathbb{E}_{S,A}^{\mathsf{hom}_{\mathbf{PSh}(\mathbf{Ord})}} \to \mathbb{E}_{S,A}^{\mathsf{hom}_{\mathbf{PSh}(\mathbf{Ord})}}$$

By Lemma 4.2.3 the morphism μ is contractive because F is locally contractive. Thus, by Theorem 4.2.6 and Remark 4.2.7 μ has a unique fixed point for which $\lfloor fix(\mu) \rfloor = \phi_A \circ F(\lfloor fix(\mu) \rfloor) \circ s^{-1}$. Hence, $\lfloor fix(\mu) \rfloor$ is the unique algebra morphism we needed. □

4.4.2 Constructing the Solution

In Section 4.4.1 we discussed that the solution to the domain equation is an F-algebra (A, ϕ_A) where ϕ_A is an isomorphism. Accordingly, our strategy to solving domain equations is to find such an F-algebra (A, ϕ_A) where ϕ_A is an isomorphism. This approach differs from the approach of Birkedal *et al.* [24] in that Birkedal *et al.* [24] work directly in the category $\mathcal{C}$ instead of $\mathcal{A}lg(F)$; see Remark 4.4.9. Although this aspect of the difference is rather superficial, it does help simplify the construction in the sense that it breaks the construction into a few simpler concepts and lemmas which are ultimately nicer to mechanize; see (canonical) partial solutions, dominating cones, *etc.* as presented below. Nevertheless, the fact that finding solutions to domain equations can be reduced to finding (initial) algebras is common knowledge [118].

Let us assume for the rest of this section that we are given a locally contractive endo-functor $F : \mathcal{C} \to \mathcal{C}$ over a **PSh**(**Ord**)-enriched category $\mathcal{C}$ which is complete, and for which all limits are enriched-pointwise. We start by defining a notion of a partial solution.

Definition 4.4.2 (Partial Solution). *Let $A \subseteq$ **Ord** be a downwards-closed subset of ordinals. An A-partial solution is an A^{op}-shaped diagram $\mathbf{P}$ in the category of F-algebras such that:*

PS-1 For any $\alpha \in A$, $\phi_{\mathbf{P}(\alpha)} : F(\mathbf{P}(\alpha)) \to \mathbf{P}(\alpha)$ is an α-isomorphism.

PS-2 For any $\beta \preceq \alpha \in A$, $\mathbf{P}_{\beta \preceq \alpha} : \mathbf{P}(\alpha) \to \mathbf{P}(\beta)$ is a β-isomorphism.

The definition of partial solutions above is local in that the conditions (PS-1) and (PS-2) only refer to individual objects or individual morphisms. As a result, given an A-partial solution $\mathbf{P}$, restricting $\mathbf{P}$ (as a diagram and hence a functor) to a downwards-closed subset $B \subseteq A$, written $\mathbf{P}|_B$, is again a B-partial solution.

Definition 4.4.3 (Dominating Cone). *Let $A \subseteq \mathbf{Ord}$ be a downwards-closed subset of ordinals and $\mathbf{P}$ an A-partial solution. We say that a cone $((D, \phi_D), \{S_\alpha\}_{\alpha \in A})$ over $\mathbf{P}$ dominates $\mathbf{P}$ if we have:*

DA-1 For any $\alpha \in A$, $S_\alpha : D \to \mathbf{P}(\alpha)$ is an α-isomorphism.

DA-2 The map ϕ_D is an α-isomorphism for any α for which we have $\{\beta | \beta \prec \alpha\} \subseteq A$.

Note that the condition (DA-2) above is equivalent to saying that ϕ_D is a $(\sup A)$-isomorphism in the event $\sup A$ does exist. In particular, the condition (DA-2) implies that if $\mathbf{P}$ is a **Ord**-partial solution, then a cone dominating it is a solution to the domain equation; see the proof of Theorem 4.4.8.

Lemma 4.4.4 (). *Let $A \subseteq \mathbf{Ord}$ be a downwards-closed subset of ordinals and $\mathbf{P}$ an A-partial solution. By Remark A.4.4 (Appendix A.4) the functor F applied to the limiting cone of $\mathbf{P}$ is also a cone on $\mathbf{P}$. We will write $DCone(\mathbf{P})$ for this cone. The cone $DCone(\mathbf{P})$ dominates $\mathbf{P}$.*

Proof. Let us write $((L, \phi_L), \{\Pi^L_\alpha\}_{\alpha \in A})$ for the cone that is the limit of $\mathbf{P}$.

First we show that ϕ_L is an A-isomorphism by showing that it is an α-isomorphism for any $\alpha \in A$. Observe that by (PS-2) we know that Corollary 4.3.14 applies and thus Π^L_α is an α-isomorphism, and by Lemma 4.3.8 so is $F(\Pi^L_\alpha)$. Furthermore, as Π^L_α is a morphism in the category of F-algebras, which means that the following diagram commutes for any $\alpha \in A$:

$$\begin{array}{ccc} F(L) & \xrightarrow{\phi_L} & A \\ {\scriptstyle F(\Pi^L_\alpha)}\downarrow & & \downarrow{\scriptstyle \Pi^L_\alpha} \\ F(\mathbf{P}(\alpha)) & \xrightarrow{\phi_{\mathbf{P}(\alpha)}} & \mathbf{P}(\alpha) \end{array}$$

Also, by condition (PS-1), $\phi_{\mathbf{P}(\alpha)}$ is an α-isomorphism. Thus, by Remark A.3.1 (Appendix A.3), ϕ_L is an α-isomorphism since the three other sides of the diagram above are α-isomorphisms.

The cone $DCone(\mathbf{P})$ that we wish to show dominates $\mathbf{P}$ is the following:

$$DCone(\mathbf{P}) = \left((F(L), F(\phi_L)), \{\phi_{\mathbf{P}(\alpha)} \circ F(\Pi^L_\alpha)\}_{\alpha \in A}\right)$$

We already established (DA-1) when we argued that the diagram above consists of α-isomorphisms. For (DA-2), let us assume we are given an ordinal α for which we have $\{\beta | \beta \prec \alpha\} \subseteq A$. We just observed that ϕ_L is an A-isomorphism and thus also a $\{\beta | \beta \prec \alpha\}$-isomorphism. Hence, by Lemma 4.3.9 $F(\phi_L)$ is an α-isomorphism. □

Next we define what we call canonical partial solutions and show how to patch canonical partial solutions together in order to construct larger ones. The latter is used in Theorem 4.4.8 for constructing partial solutions by well-founded induction on ordinals.

Definition 4.4.5 (Canonical Partial Solutions). *Let $A \subseteq \mathbf{Ord}$ be a downwards-closed subset of ordinals and $\mathbf{P}$ an A-partial solution. We say $\mathbf{P}$ is a canonical partial solution if it is constructed at all stages via the construction in Lemma 4.4.4. That is, if for any $\alpha \in A$ we have*

$$\left(\mathbf{P}(\alpha), \{\mathbf{P}_{\beta \preceq \alpha}\}_{\beta \preceq \alpha}\right) = DCone\left(\mathbf{P}|_{\{\beta | \beta \prec \alpha\}}\right)$$

are equal cones of the diagram $\mathbf{P}|_{\{\beta|\beta\prec\alpha\}}$.

Lemma 4.4.6 ( ). *Let $\mathbf{P}$ and $\mathbf{Q}$ be two canonical A-partial solutions. We have $\mathbf{P} = \mathbf{Q}$ (as diagrams,* i.e.*, functors).*

On paper, the Lemma 4.4.6 above is proven through a simple argument by transfinite induction. However, as we will discuss in Section 4.5, it is far from obvious to mechanize.

Lemma 4.4.7 ( ). *Let $\{\mathbf{P}^\alpha\}_{\alpha\in A}$ be a collection of canonical partial solutions indexed by some downwards-closed subset of ordinals A such that $\mathbf{P}^\alpha$ is a canonical $\{\beta|\beta \preceq \alpha\}$-partial solution. We can construct a canonical A-partial solution $\mathbf{Q}$ by patching the partial solutions $\{\mathbf{P}^\alpha\}_{\alpha\in A}$ together. That is, we take $\mathbf{Q}(\alpha) := \mathbf{P}^\alpha(\alpha)$ and take $\mathbf{Q}_{\beta\preceq\alpha} := \mathbf{P}^\alpha_{\beta\preceq\alpha}$.*

Note that the proof, and even the well-formedness of the statement of Lemma 4.4.7 above depends on Lemma 4.4.6. In particular, note that $\mathbf{Q}_{\beta\preceq\alpha}$ must be a morphism from $\mathbf{Q}(\alpha)$ to $\mathbf{Q}(\beta)$, or equivalently from $\mathbf{P}^\alpha(\alpha)$ to $\mathbf{P}^\beta(\beta)$, whereas the morphism $\mathbf{P}^\alpha_{\beta\preceq\alpha}$ is a morphism from $\mathbf{P}^\alpha(\alpha)$ to $\mathbf{P}^\alpha(\beta)$. Thus, one would need to prove $\mathbf{P}^\alpha(\beta) = \mathbf{P}^\beta(\beta)$ for it to even make sense to take $\mathbf{Q}_{\beta\preceq\alpha} := \mathbf{P}^\alpha_{\beta\preceq\alpha}$. This is the case because by Lemma 4.4.6 $\mathbf{P}^\alpha|_{\{\gamma|\gamma\preceq\beta\}} = \mathbf{P}^\beta$. However, in type theory, in our Rocq mechanization, one needs to work up to the equality $\mathbf{P}^\alpha(\beta) = \mathbf{P}^\beta(\beta)$ (transport along this equality) when defining $\mathbf{Q}$, which also includes establishing its functoriality, that it is a partial solution, and its canonicity. We will discuss these subtleties in Section 4.5.

Theorem 4.4.8 ( ). *The locally contractive functor F has a solution.*

Proof. We first construct a canonical $\mathbf{Ord}$-partial solution $\mathbf{P}$. By Lemma 4.4.7 it suffices to construct canonical $\{\beta|\beta\preceq\alpha\}$-partial solutions for all $\alpha \in \mathbf{Ord}$. We do so by well-founded induction on α. Thus, let us assume that we have canonical $\{\gamma|\gamma\preceq\beta\}$-partial solutions for all $\beta \prec \alpha$. We use Lemma 4.4.7 to construct a canonical $\{\beta|\beta\preceq\alpha\}$-partial solution as required. The solution is the F-algebra of $DCone(\mathbf{P})$. We only need to show that the map $\phi_{DCone(\mathbf{P})}$ is an isomorphism. By

Lemma 4.3.7 it suffices to show that $\phi_{DCone(\mathbf{P})}$ is an α-isomorphism for all $\alpha \in \mathbf{Ord}$. However, by Lemma 4.4.4 $DCone(\mathbf{P})$ dominates $\mathbf{P}$. Hence, by the property (DA-2) of dominating cones we only need to show that $\{\beta | \beta \prec \alpha\} \subseteq \mathbf{Ord}$, which holds trivially. □

Remark 4.4.9. *In addition to the difference of working with the category of F-algebras as opposed to working directly in $\mathcal{C}$, our approach to solving domain equations differs from that presented by Birkedal* et al. *[24] in how we treat zero and limit ordinals. Working with sheaves, Birkedal* et al. *[24] at zero and limit ordinals simply take the limit of the construction at stages below. By contrast, we apply F to the obtained cone of the limit at every single stage of the construction and not just at successor ordinals. Another way to look at this difference is if we look at the sequence of objects constructed in these two approaches (in our case the carrier objects of the algebras we compute). Up to isomorphism, what we compute is the sequence X below while what Birkedal* et al. *[24] compute is the sequence Y:*

$$\begin{array}{lllllll} X_0 := F(1); & X_1 := F(F(1)); & X_2 := F(F(F(1))); & \cdots & X_\omega := F(\lim_{\alpha \prec \omega} X_\alpha); & X_{\omega^+} := F(X_\omega); & \cdots \\ Y_0 := 1; & Y_1 := F(1); & Y_2 := F(F(1)); & \cdots & Y_\omega := \lim_{\alpha \prec \omega} Y_\alpha; & Y_{\omega^+} := F(Y_\omega); & \cdots \end{array}$$

4.4.3 Mixed-Variance Domain Equations

In addition to covariant functors of the form $\mathcal{C} \to \mathcal{C}$, we in general need to [24] solve domain equations for mixed-variance functors of the form $\mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$.

Example 4.4.10 (🚒). *As a simple minimal example we have used our development to solve the domain equation for the following mixed-variance functor which is a simplified version of the functor used by Frumin* et al. *[49]:*

$$F(X,Y) := \Delta(\mathbb{N}) + \blacktriangleright(Y^X) + \blacktriangleright Y$$

where $\Delta(A)$ is the constant presheaf mapping all ordinals to the set A.

The following lemma shows that *mixed-variance* locally contractive functors $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$ like the one in Example 4.4.10 have unique solutions.

Lemma 4.4.11 (🚒). *Let $\mathcal{C}$ be a* **PSh**(**Ord**)*-enriched category and $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$ be a locally contractive functor (if $\mathcal{C}$ is enriched, so is $\mathcal{C}^{\mathsf{op}}$ and also their product). Furthermore, assume that $\mathcal{C}$ is complete and co-complete with enriched-pointwise limits and co-limits. The functor F has a unique solution.*

Proof. Define the functor $\tilde{F}(A,B) := (F(B,A), F(A,B))$ from $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$ to $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$. The functor $\tilde{F}$ is also locally contractive and hence, by Theorem 4.4.8, has a solution, say (X,Y) for which $\tilde{F}(X,Y) \simeq (X,Y)$. In that case, by symmetry of $\tilde{F}$, we have $\tilde{F}(Y,X) \simeq (Y,X)$. Thus, by Theorem 4.4.1, $(X,Y) \simeq (Y,X)$ which implies that $X \simeq Y$, and hence, $F(X,X) \simeq X$. □

4.5 Rocq Mechanization

All results that have been marked by a symbol throughout the paper and the appendices are mechanized [121] in the Rocq Prover. For this mechanization we have used the step-indexing interface of Spies *et al*. [119] which abstracts a step-indexing structure that Spies *et al*. [119] instantiate twice: once with natural numbers in Rocq (for step-indexing over ω), and once with ordinal numbers. Spies *et al*. [119] use the mechanization of ordinal numbers by Kirst *et al*. [76]. We use the following axioms in our mechanization: axiom of choice (and its consequence excluded middle), propositional extensionality (and its consequences proof irrelevance and uniqueness of identity proofs (UIP)), and functional extensionality. The first two of these axioms are already assumed by Kirst *et al*. [76] to construct a model of set theory in Rocq. The inclusion of functional extensionality, on the other hand, is necessary for formalization of category theory, at least the way we have; we will discuss this below.

The Notion of Equality of Homorphisms in Type Theory There are many efforts mechanizing category theory in type-theory-based proof assistants [3, 55, 59, 60, 89, 94, 131, 134, 141]. Hu and Carette [59] give an extensive survey comparing existing type-theory-based category theory mechanizations across multiple different axes. One important design point in mechanizing category theory in type theory is representing equality of morphisms. Roughly speaking, this design decision divides mechanizations into two camps: those using setoids, also known as Bishop sets [26], for representing morphisms [59, 60, 94, 141] and those using equality [3, 55, 89, 131, 134] (including those working in HoTT [134] settings where equality plays a key role).

There are multiple advantages to using setoids as pointed out by Hu and Carette [59]. Hu and Carette [59] work in Agda and state "Our principal theoretical contribution is to show that *setoid-based proof-relevant category theory* works just as well as various other 'flavours' of category theory by supporting a large number of definitions and theorems." One of the main advantages of using setoids is avoiding axioms such as functional extensionality (for proving equality of functors, natural transformations, *etc*.) and classical axioms (for constructing quotient types, *e.g*., for co-limits in **Set** or presheaf categories) — of course, works based in HoTT admit these as theorems. Indeed, our development also started with implementing the necessary basic concepts in category theory *using setoids*. However, we discovered an issue that fundamentally precludes the use of setoids for morphisms in our mechanization. This problem arises in the proof of Lemma 4.4.6. For this lemma, we need to show that two F-algebras are equal which are constructed based on the limits of two diagrams that are equal (by our transfinite induction hypothesis). However, the natural notion of equality of functors in setoid settings is to ask that the morphism maps of functors map setoid-equal morphisms into setoid-equal morphisms. Consequently, the best one can prove is that equal functors in this sense produce isomorphic limits. Thus, instead of Lemma 4.4.6, one could prove that canonical partial solutions are naturally isomorphic. However, as remarked after Lemma 4.4.7 in Section 4.4.2, this is not even sufficient for the

statement of Lemma 4.4.7 to be well-formed. This is why we chose to change our mechanization to use the equality notion from Rocq's standard library instead of setoid equality. It appears that existing mechanizations of category theory that formalize collections of morphisms as setoids have never attempted formalizing a construction such as ours that involves defining a functor on ordinals by transfinite induction where the construction at each stage involves taking the limit of the construction up to below that stage.

Working With All Ordinals in The Universe As we discussed in Section 4.1, we work with the category of presheaves over all ordinals in the universe, *i.e.*, the set **Ord** which is *not* closed under suprema. Thus, many of our definitions are parameterized by a downwards-closed subset of ordinals. The collection of downwards-closed subsets of ordinals, ordered by the subset relation, can be thought of as the ideal completion of **Ord**. In particular, the total set **Ord** is the maximal element of this order which represents the supremum of the entire set **Ord**. This means that the key lemma of our formalization, Lemma 4.4.7, is applicable to both proper downwards-closed subsets of ordinals (which represent ordinals that do happen to be in **Ord**) as well as the aforementioned supremum of the entire set **Ord** (which itself is not in **Ord**). This is why we can apply Lemma 4.4.7 twice in the proof of Theorem 4.4.8, once for constructing $\{\beta | \beta \preceq \alpha\}$-partial solutions for all $\alpha \in \mathbf{Ord}$ by transfinite induction on α, and once to put all those together to construct a **Ord**-partial solution. This is key in significantly reducing the size of the mechanization as otherwise a mechanization working with all ordinals in the universe would have to prove two different versions of Lemma 4.4.7 for the two different use cases in Theorem 4.4.8; once for individual ordinals, and once for all ordinals. Spies *et al.* [119] also work with the collection of ordinals in the universe; as mentioned above we took their notion of step-indexing and ordinals verbatim in our mechanization. And, they indeed duplicate Lemma 4.4.7 as we just explained. This requires them to repeat multiple definitions and lemmas for these two versions of our single Lemma 4.4.7.

Well-Behaved Subset Types in Rocq Working with downwards-closed subsets of ordinals, we need to formalize them in our Rocq mechanization. In our mechanization, we define downwards-closed subsets of ordinals as *subset types* which we represent as a record that packages an ordinal together with a *proof* that said ordinal is in the downwards-closed subset. We first define downwards-closed predicates over ordinals, `downset_pred`, as a record type consisting of a predicate together with a proof that this predicate is downwards-closed (ignore the decidability part for now, we will get back to it):

```
Polymorphic Record downset_pred (SI : indexT) := MkDownSetPred {
  dsp_pred :> SI → Prop;
  dsp_pred_dec : ∀ α, Decision (dsp_pred α);
  dsp_pred_downwards : ∀ α β, α ≼ β → dsp_pred β → dsp_pred α;
}.
```

Here, `indexT` is the generalized type exposed by the step-indexing interface of Spies *et al.* [119]; the type of all ordinals in the universe being an instance of this structure. Based on the downwards-closed predicates defined above, we would like to define downwards-closed subsets essentially as a record type that packages together an ordinal, together with a proof that it belongs to the provided downwards-closed predicate. However, a naïve encoding as such a record type leads to a problem: given a downwards-closed predicate over ordinals, two elements of such a type with the same ordinals but different proofs would not be *definitionally* equal — of course, they are *propositionally* equal as we assume proof irrelevance. This problem is especially noticeable when we look at two downwards-closed subsets where one is included in the other. We define the inclusion relation between two downwards-closed predicates `dsp` and `dsp'` as one would expect: `∀ α, dsp α → dsp' α`. This allows us to define a simple function that lifts ordinals from the smaller downwards-closed subset to the larger one. We use this function in our Rocq mechanization to define the restriction operation in Section 4.4.2 on presheaves over downwards-closed subsets of ordinals.

We solve the issue discussed above by defining the record type `downset` as follows:

```
#[projections(primitive = yes)]
Record downset {SI} (dsp : downset_pred SI) := MkDS {
  ds_idx :> SI;
  ds_in_dsp : squashed (dsp ds_idx);
}.
```

where the type `squashed` is exactly as defined in Gilbert *et al.* [53]:

```
Inductive squashed (P : Prop) : SProp := squash : P → squashed P.
```

The idea here is that since `squashed` is in the universe `SProp` of definitionally proof-irrelevant propositions, and the fact that type `downset` is defined as a record with primitive projections (and hence it is subject to the η conversion law for records), two terms of the type `downset dsp` are *definitionally* equal as soon as their underlying ordinal, the projection `ds_idx`, are *definitionally* equal. Now, the problem is that when working with elements of `downset dsp`, we need to have a proof that the underlying ordinal is indeed in `dsp`, *i.e.*, we need something of type `dsp ds_idx`, whereas we are *only* given a proof of `squashed (dsp ds_idx)`. Importantly, the type `squashed` above cannot be eliminated to produce a term of a type that is outside the universe `SProp` — in technical terms, this is because the first argument of the constructor `squash` (the argument with type `P`) is *non-forced*, and is also not in `SProp` [53]. Nevertheless, inspired by the constant map from identity proofs to identity proofs in the proof of Hedberg's theorem [58], we prove the following lemma which allows us to recover a proof of `dsp ds_idx` from an element of `squashed (dsp ds_idx)`:[4]

```
Lemma unsquash {P : Prop} `{!Decision P} (s : squashed P) : P.
```

The lemma above is the reason why we included a proof of decidability of the subset predicate in the definition of `downset_pred` above.

[4] Gilbert *et al.* [53] use the name `unsquash` for the eliminator of their `squashed` type (which they call `squash`, and its constructor `sq`) that only eliminates into other `SProp` types.

4.6 Related Works

Domain Theory We have already discussed works on domain theory that are most closely related to ours in Section 4.1, including the most closely related work to ours [24] which our work is closely based on, and which we have compared our work to throughout the paper.

Fixed Points in Type Theory When working with inductive and co-inductive types and proofs in type theory, it is required to follow restrictive syntactic checks (*e.g.*, productivity and guardedness for co-induction). These overly strong syntactic conditions protect mechanizations against inconsistencies, but reject many valid definitions. Motivated alleviate this situation, Di Gianantonio and Miculan [51] introduce complete ordered families of equivalences (COFEs) as a unifying theory for mixed-variance recursive definitions that support construction of fixed points. They define COFEs over an arbitrary well-founded order and prove a generalized fixed point theorem for contractive endofunctions over these COFEs. In a subsequent work, Di Gianantonio and Miculan [52] generalize this result to sheaf categories over topologies with a well-founded basis — this is very close to the setting of Birkedal *et al*. [24] upon which we have based our work. The main difference between the works of Di Gianantonio and Miculan [51, 52] and Birkedal *et al*. [24], and thereby also our work, is that the former only constructs fixed points of morphisms (similar to our results in Section 4.2.2) whereas the latter also constructs fixed points of functors.

Mechanizations of Solutions to Domain Equations Benton *et al*. [15] mechanize solution to domain equations over directed-complete partial orders (DCPOs) in the Rocq Prover based on the mechanization of DCPOs by Paulin-Mohrig [92]. Huffman [61] constructs a universal domain into which all bifinite domains can be embedded. Dockins [35] mechanizes solutions of domain equations over the category of profinite domains [56] in Rocq. All these works are based on classical domain theory, and as also pointed out by Sieczkowski *et al*. [112], unlike our guarded domains, do not appear to be suitable for modeling higher-order program logics like Iris [73].

The most closely related works to us are Rocq mechanizations of the domain equation solver of the ModuRes library [112], the domain equation solver of the Iris program logic [72] which is a nicer reimplementation of the domain equation solver of the ModuRes library, and the domain equation solver of transfinite Iris [119]. The former two mechanizations work with the category of COFEs (these are COFEs over ω and not over an arbitrary ordered set like Di Gianantonio and Miculan [51]), a representation of the category of complete bisected bounded ultra metric spaces (CBUlt) [22] that is particularly amenable to mechanizations [112]. These works only support step-indexing up to ω. Transfinite Iris, inspired by Birkedal *et al*. [24], extend the definition of OFEs (COFEs without completeness requirement) and COFEs over ω to those over **Ord**. However, Transfinite Iris, unlike the ModuRes library and Iris, only solves domain equations for functors of the form $\mathrm{OFE}^{\mathrm{op}} \times \mathrm{OFE} \to \mathrm{COFE}$ and

not $\mathrm{COFE}^{\mathrm{op}} \times \mathrm{COFE} \to \mathrm{COFE}$. An example of a functor that is not supported by transfinite Iris as a result of this limitation is our Example 4.4.10.

Mechanizations of Category Theory We mentioned the existing mechanizations of category theory in Section 4.5. We refer to Hu and Carette [59] who give an extensive survey comparing these mechanizations. Regarding our mechanization of category theory, we only mention that its span is not significant compared to the works cited, compared, and contrasted by Hu and Carette [59]. We have only mechanized what was necessary for formalizing our main results: Theorem 4.4.1, Theorem 4.4.8 and Lemma 4.4.11.

4.7 Future Work

Our main future goal is to build a step-indexed (program) logic similar to the Iris framework [73] based on our development. We hope to use such a step-indexed logic to study weak bisimulation of guarded interaction trees, i.e., objects similar to the one shown in Example 4.4.10. This requires transfinite step-indexing because we need to allow either side of the bisimulation relation to take finitely many silent steps (τ-steps). However, as we discussed in Section 4.6, the existing work on transfinite step-indexing does not support equations like that in Example 4.4.10. However, there is a significant amount of nontrivial technical work that needs to be done before we can put our mechanization to use for constructing a user-friendly step-indexed (program) logic framework, *e.g.*, providing an interactive proof mode similar to that of the Iris framework [79, 80]. In principle, the main limiting factor is the complexity of working with presheaves compared to COFEs in proof assistants; using our solution forces one to always use categories and categorical constructions. For instance, maps between COFEs are non-expansive functions (Rocq functions with a side-condition), while maps between presheaves are natural transformations (families of functions with a naturality side-condition relating these families). Thus, while presheaves are more amenable to mechanization than sheaves, there remains a substantial amount of work required to build a user-friendly (program) logic framework on top of our category theoretic development. This makes developing a user-friendly system on top of our development very challenging, which we leave as an important future work.

4.8 Conclusion

After motivating the need for solving domain equations over the category of presheaves over ordinals, we presented the theory of solving such domain equations and discussed its mechanization as well as the challenges we faced mechanizing this theory. As demonstrated by our Example 4.4.10, this domain equation solver can be used to solve domain equations stated as mixed-variance functors like those that are needed for guarded interaction trees [49] or program logics [119].

Part III

Appendix

APPENDIX A

Solving Guarded Domain Equations in Presheaves Over Ordinals and Mechanizing It

A.1 The Need for Step-Indexing Over Higher Ordinals

As we discussed in Section 4.1, the main issue is that when working with step-indexing over ω the existential property fails to hold. That is, if we know that $\models \exists x : A.\ \phi(x)$ holds, we do necessarily get that there exists an $a \in A$ such that $\models \phi(a)$ holds. In this appendix, we first discuss why the existential property fails when step-indexing over ω. We then present a proof for a general criterion of when the existential property does hold, and based on this criterion motivate our choice (and that of Spies *et al.* [119]) to use step-indexing over all ordinals in a Grothendieck universe.

A.1.1 Failure of the Existential Property

Recall that in step-indexing over ω the set of truth values is the Heyting algebra of the downwards-closed subsets of ω. In this setting, the interpretation of ϕ, $[\![\phi]\!]$, is a function from A into this Heyting algebra, and we have $[\![\exists x : A.\ \phi(x)]\!] = \bigcup_{x \in A} [\![\phi]\!](x)$. Thus, $\models \exists x : A.\ \phi(x)$ is equivalent to saying that $\bigcup_{x \in A} [\![\phi]\!](x) = \omega$, *i.e.*, that the interpretation of $\exists x : A.\ \phi(x)$ is the truth value $\top$, which in our Heyting algebra is the entire set ω. Now, take $A := \mathbb{N}$ and $\phi(n) := \triangleright^n \bot$. Readers not familiar with step-indexed logics can ignore the exact definition of ϕ. (These readers are kindly referred to Jung *et al.* [73] for a detailed explanation of the syntax and semantics of the step-indexed logic Iris). What is important for us is that $[\![\triangleright^n \bot]\!] = \{k | k < n\}$. Based on this interpretation one can easily see that $\bigcup_{n \in \mathbb{N}} [\![\triangleright^n \bot]\!] = \omega$ but there is no $n \in \mathbb{N}$ such that $[\![\triangleright^n \bot]\!] = \omega$.

A.1.2 A General Criterion for the Existential Property to Hold

Here, we present a general criterion of when the universal property holds. This section is based on a blog post by the second author [129].

Theorem A.1.1 (). *If the step-indexing is a regular ordinal and the cardinality of the set quantified over is strictly smaller than that of the step-indexing ordinal, then the existential property holds.*

Remark A.1.2. *The Rocq mechanization of Theorem A.1.1 does not use ordinals or a step-indexed logic. It follows very closely the ideas in the blog post [129]. It first defines an analogue of regularity for an arbitrary Rocq type A with a relation $R \subseteq A \times A$ on it. Then, it shows that $\forall a : A.\ \exists b : B.\ P(a,b)$ implies $\exists b : B.\ \forall a : A.\ P(a,b)$ whenever the type A is regular, the cardinality of B is strictly smaller than that of A, and furthermore P is downwards-closed with respect to R,* i.e., *if $\forall a, a' \in A, b \in B.\ R(a,a') \wedge P(a',b) \implies P(a,b)$.*

Below, we first give the definition of regular ordinals (which can be found in most standard textbooks on set theory [64]) and then give the proof of Theorem A.1.1.

Definition A.1.3. *We say an ordinal γ is regular, if the supremum of any sequence of ordinals strictly smaller than γ, indexed by an ordinal strictly smaller than γ, is also strictly smaller than γ. In other words, for any sequence of ordinals $\{\beta_\alpha\}_{\alpha \preceq \delta}$ indexed by an ordinal δ, if we have both that $\delta \prec \gamma$, and that $\forall \alpha \preceq \delta.\ \beta_\alpha \prec \gamma$, then $\bigcup_{\alpha \preceq \delta} \beta_\alpha \prec \gamma$.*

Proof of Theorem A.1.1. Let us assume that we are working with a logic step-indexed over a regular ordinal γ — thus, the set of truth values is the Heyting algebra of downwards-closed subsets of γ. Furthermore, let us assume we are given a set A whose cardinality is strictly smaller than that of γ. (More specifically, let us assume that $A = \{a_\alpha | \alpha \preceq \delta\}$ for some $\delta \prec \gamma$.) Finally, assume we are given a predicate ϕ over A such that $[\![\exists x : A.\ \phi(x)]\!] = \bigcup_{x \in A} [\![\phi]\!](x) = \gamma$. We show, using proof by contradiction, that there exists an element $a \in A$ such that $[\![\phi]\!](a) = \gamma$.

Assume, to the contrary that there is no element $a \in A$ such that $[\![\phi]\!](a) = \gamma$. In other words, for any $a \in A$ there is an ordinal $\beta \prec \gamma$ such that $\beta \notin [\![\phi]\!](a)$. Now, since $A = \{a_\alpha | \alpha \preceq \delta\}$, this forms a sequence of ordinals $\{\beta_\alpha\}_{\alpha \preceq \delta}$ such that $\forall \alpha \preceq \delta.\ \beta_\alpha \prec \gamma$ and that $\forall \alpha \preceq \delta.\ \beta_\alpha \notin [\![\phi]\!](a_\alpha)$. Thus, by regularity of γ, we have that $\bigcup_{\alpha \preceq \delta} \beta_\alpha \prec \gamma$, and by the fact that for any a the set $[\![\phi]\!](a)$ is downwards-closed, we have that $\forall \alpha \preceq \delta.\ \bigcup_{\alpha \preceq \delta} \beta_\alpha \notin [\![\phi]\!](a_\alpha)$. Hence, it must be the case that $\bigcup_{\alpha \preceq \delta} \beta_\alpha \notin \bigcup_{\alpha \preceq \delta} [\![\phi]\!](a_\alpha) = \bigcup_{x \in A} [\![\phi]\!](x) = [\![\exists x : A.\ \phi(x)]\!]$, which is a contradiction. □

We remark that the set of all ordinals in the universe acts basically as a regular ordinal — indeed, this must be understood as step-indexing over the supremum of that set which is regular. In other words, we have by definition that for any function $A \to \mathbf{Ord}$ where A is a set/type in the universe, the supremum of the image of the function is again an ordinal in **Ord**. Thus, when step-indexing over all ordinals

in the universe, by Theorem A.1.1, we get that the existential property holds for quantification over any set in the universe.

A.2 Later is Locally Contractive, Earlier is not Even Enriched

Theorem A.2.1 (Later is Enriched and Locally Contractive). *The functor* $\blacktriangleright : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$ *is an enriched and locally contractive functor.*

Remark A.2.2 (Earlier is *not* Enriched)**.** *The category* $\mathbf{PSh}(\mathbf{Ord})$ *is enriched over itself. Hence, we can ask whether the functor* $\blacktriangleleft : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$ *is an enriched functor? The answer is negative. Here we give an intuitive explanation as to why this is not the case. We will formally prove this negative answer in Lemma A.2.3.*

To understand why earlier is not enriched note that we need to produce a natural transformation for the internal action of the earlier functor. That is, a natural transformation $G^F \to \blacktriangleleft G^{\blacktriangleleft F}$*. By adjunction of exponentiation it is the same as requiring a natural transformation* $G^F \times \blacktriangleleft F \to \blacktriangleleft G$*. Let us look at an arbitrary component of this natural transformation at stage* α*. That is, a function (morphism in* **Set***)* $(\mathcal{Y}_\alpha \times F \to G) \times F(\alpha^+) \to G(\alpha^+)$*. Such a map, given a natural transformation* $\eta : \mathcal{Y}_\alpha \times F \to G$ *and an element* $x \in F(\alpha^+)$ *must produce a an element of* $G(\alpha^+)$*. The only possibility here is to use* $\eta_{\alpha^+} : \mathcal{Y}_\alpha(\alpha^+) \times F(\alpha^+) \to G(\alpha^+)$*. However, set* $\mathcal{Y}_\alpha(\alpha^+) = \{*|\alpha^+ \preceq \alpha\}$ *is empty.*

As a simple corollary of uniqueness of solutions for locally contractive functors we prove that the functor earlier cannot be an enriched functor.

Lemma A.2.3. *The functor* $\blacktriangleleft : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$ *is not an enriched functor for the self-enrichment of the category* $\mathbf{PSh}(\mathbf{Ord})$*.*

Proof. Assume that $\blacktriangleleft$ is an enriched functor. In that case, by Lemma 4.3.4 the functor $\blacktriangleleft \circ \blacktriangleright$ would be a locally contractive functor as $\blacktriangleright$ is by Lemma A.2.1. However, the functor $\blacktriangleleft \circ \blacktriangleright$ is naturally isomorphic to $\mathsf{id}_{\mathbf{PSh}(\mathbf{Ord})}$. Recall that the co-unit of the adjunction $\blacktriangleleft \dashv \blacktriangleright$ is a natural isomorphism. Hence, any presheaf is a solution for the locally contractive functor $\blacktriangleleft \circ \blacktriangleright$, *i.e.*, for any presheaf F we have $\blacktriangleleft(\blacktriangleright F) \simeq F$. Consequently, by Lemma 4.4.1 all presheaves over ordinals must be isomorphic which is a contradiction. □

An alternative proof could be given through violating Lemma 4.3.8. Consider the unique morphism $f : \blacktriangleright 0 \to 1$ (0 and 1 being the initial and terminal presheaf respectively). This morphism is a 0-isomorphism while $\blacktriangleleft f : 0 \to 1$ (note that $\blacktriangleleft \blacktriangleright 0 = 0$ and $\blacktriangleleft 1 = 1$) is not a 0-isomorphism.

A.3 Omitted Properties of Ordinal-Partial Isomorphisms

Remark A.3.1 ( ). *Ordinal-partial isomorphisms satisfy many properties that ordinary isomorphisms do. In particular, we remark the properties listed below which are all easy to show:*

PIso-1 *Identity morphisms* id_A *are* α*-isomorphisms for any* α*.*

PIso-2 *For any* α*-isomorphism* $f : A \to B$ *and any* $g, h \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{B,C}(\alpha)$*:*

$$\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{A,B,C}(\alpha)(\lceil f \rceil(\alpha)(*), g) = \mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{A,B,C}(\alpha)(\lceil f \rceil(\alpha)(*), h) \implies g = h$$

PIso-3 *For any* α*-isomorphism* $f : A \to B$ *and any* $g, h \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{C,A}(\alpha)$*:*

$$\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{C,A,B}(\alpha)(g, \lceil f \rceil(\alpha)(*)) = \mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{C,A,B}(\alpha)(h, \lceil f \rceil(\alpha)(*)) \implies g = h$$

PIso-4 *For any* α*-isomorphism* $f : A \to B$ *where* $x \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{B,A}(\alpha)$ *is* f*'s partial inverse and any* $g, h \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{A,C}(\alpha)$*:*

$$\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{B,A,C}(\alpha)(x, g) = \mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{B,A,C}(\alpha)(x, h) \implies g = h$$

PIso-5 *For any* α*-isomorphism* $f : A \to B$ *where* $x \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{B,A}(\alpha)$ *is* f*'s partial inverse and any* $g, h \in \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{C,B}(\alpha)$

$$\mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{C,B,A}(\alpha)(g, x) = \mathbb{E}^{\mathsf{comp}_{\mathcal{C}}}_{C,B,A}(\alpha)(h, x) \implies g = h$$

PIso-6 *If* $f : A \to B$ *and* $g : B \to C$ *are both* α*-isomorphisms then so is* $g \circ f$*.*

PIso-7 *For any* $f : A \to B$ *and* $g : B \to C$*, if* $g \circ f$ *is an* α*-isomorphism, then* f *is an* α*-isomorphism if and only if* g *is an* α*-isomorphism.*

A.4 Some Categorical Definitions and Constructions

Here we present a few basic and well-known category theoretic facts and constructions that are nevertheless worth presenting here so that we can refer to them in the main text.

A.4.1 Some Properties of Presheaves Over Ordinals

Lemma A.4.1 ( ). *Let* F *be a presheaf over ordinals and* $\gamma \preceq \alpha$ *be two ordinal numbers. The set* $F(\alpha)$ *is the limit of the diagram* $F|_{\{\beta \mid \gamma \preceq \beta \preceq \alpha\}}$ *being the functor* F *where the domain is restricted to the set of ordinals* $\{\beta \mid \gamma \preceq \beta \preceq \alpha\}$*.*

A.4.2 Extending Partial Ordinal-Shaped Diagrams

Here, by a partial ordinal-shaped diagram we mean diagram $F : \{\beta | \beta \prec \alpha\}^{\mathsf{op}} \to \mathcal{C}$ whose index category is ordinals strictly below a certain ordinal α. We show that given a cone $(V, \{S_\gamma\}_{\gamma \in \{\beta | \beta \prec \alpha\}})$ on F, we can extend the diagram into a diagram whose index is $\{\beta | \beta \preceq \alpha\}^{\mathsf{op}}$. We write $F^{\mathsf{Ext}(V)}$ for this extended diagram.

$$F^{\mathsf{Ext}(V)}(\gamma) = \begin{cases} V & \text{if } \gamma = \alpha \\ F(\gamma) & \text{otherwise} \end{cases}$$

$$F^{\mathsf{Ext}(V)}_{\delta \preceq \gamma} = \begin{cases} \mathsf{id}_V & \text{if } \gamma = \alpha \text{ and } \delta = \alpha \\ S_\delta & \text{if } \gamma = \alpha \text{ and } \delta \prec \alpha \\ F_{\delta \preceq \gamma} & \text{otherwise} \end{cases}$$

The fact that $(V, \{S_\gamma\}_{\gamma \in \{\beta | \beta \prec \alpha\}})$ is a cone on F ensures that $F^{\mathsf{Ext}(V)}$ is a functor and hence a $\{\beta | \beta \preceq \alpha\}^{\mathsf{op}}$-shaped diagram.

A.4.3 (Co-)Limits in Functor Categories are Pointwise

Consider the category of functors from $\mathcal{C}$ to $\mathcal{D}$ and natural transformation between them. It is well known that the category $\mathcal{D}^{\mathcal{C}}$ is complete whenever $\mathcal{D}$ is. Furthermore, in that case limits are pointwise. We state this fact formally here. Consider a diagram $F : \mathcal{J} \to \mathcal{D}^{\mathcal{C}}$. Given an object A of $\mathcal{C}$, we define the pointwise diagram $F^A : \mathcal{J} \to \mathcal{D}$ as the functor defined as follows:

$$\begin{aligned} F^A(J) &:= F(J)(A) \\ F^A(h) &:= F(h)(\mathsf{id}_A) \qquad\qquad \text{for any morphism } h : J \to J' \end{aligned}$$

Theorem A.4.2 (Limits in Functor Categories). *Given a diagram $F : \mathcal{J} \to \mathcal{D}^{\mathcal{C}}$, a functor $L : \mathcal{C} \to \mathcal{D}$ is the limit of the diagram F if and only if for any object A of $\mathcal{C}$, $L(A)$ is the limit of the diagram F^A.*

Similarly, co-limits in functor categories are pointwise — the functor category is co-complete whenever the co-domain category is.

A.4.4 On Algebras of Endo-Functors and their Categories

Recall that given an endo-functor $T : \mathcal{C} \to \mathcal{C}$, a T-algebra is a pair (A, ϕ_A) of an object A of $\mathcal{C}$ together with a morphism $\phi_A : T(A) \to A$. Furthermore, an algebra morphism from (A, ϕ_A) to (B, ϕ_B) is a morphism $h : A \to B$ such that the following diagram commutes:

$$\begin{array}{ccc} T(A) & \xrightarrow{\phi_A} & A \\ {\scriptstyle T(h)}\downarrow & & \downarrow{\scriptstyle h} \\ T(B) & \xrightarrow{\phi_B} & B \end{array} \qquad \text{(alg-hom)}$$

For any endo-functor T, T-algebras together with T-algebra morphisms form a category $\mathcal{A}lg(T)$. We write $\mathcal{U}_{\mathcal{A}lg(T)} : \mathcal{A}lg(T) \to \mathcal{C}$ for the forgetful functor of $\mathcal{A}lg(T)$.

Theorem A.4.3 (). *Let T be an endo-functor on $\mathcal{C}$. The category $\mathcal{A}lg(T)$ is complete whenever $\mathcal{C}$ is.*

Proof. Let $F : \mathcal{J} \to \mathcal{A}lg(T)$ be a diagram of T-algebras. We endow the limit of the $\mathcal{C}$ diagram $\mathcal{U}_{\mathcal{A}lg(T)} \circ F$ with an algebra structure. Let L be the limit of this diagram with projections $\Pi^L_J : L \to F(J)$. The morphisms $\phi_{F(J)} \circ T(\Pi^L_J) : T(L) \to F(J)$ form a cone on the diagram $\mathcal{U}_{\mathcal{A}lg(T)} \circ F$. We define $\phi_L : T(L) \to L$ as the unique morphism into the limit L from this cone. That is,

$$\phi_L := \lim_{J \in \mathcal{J}} \left(\phi_{F(J)} \circ T(\Pi^L_J) \right)$$

It remains to show that the projections of the limit $\Pi^L_J : L \to F(J)$ are algebra homomorphisms, and that given any cone over F in the category $\mathcal{A}lg(T)$ there is a unique T-algebra homomorphism from that cone to (L, ϕ_L). We leave the latter as a simple exercise. As for the former, we need to show that the following diagram commutes

$$\begin{array}{ccc} T(L) & \xrightarrow{\phi_L} & A \\ {\scriptstyle T(\Pi^L_J)}\downarrow & & \downarrow{\scriptstyle \Pi^L_J} \\ T(F(J)) & \xrightarrow{\phi_{F(J)}} & F(J) \end{array}$$

which simply holds by the definition of ϕ_L above. □

Remark A.4.4 (). *For any T-algebra (A, ϕ_A), $(T(A), T(\phi_A))$ is also a T-algebra. Furthermore, if h is a T-algebra morphism, so is $T(h)$. Thus, T forms an endo-functor on the category of T-algebras. Consequently, the image of any commutative diagram in $\mathcal{A}lg(T)$ under T is also a commutative diagram. Hence, for a diagram $F : \mathcal{J} \to \mathcal{A}lg(T)$ of T-algebras and a cone $\left((A, \phi_A), \{ S_j : A \to F(J) \}_{j \in J} \right)$ on diagram F, the cone below is also a cone on diagram F:*

$$\left((T(A), T(\phi_A)), \{ \phi_{F(J)} \circ T(S_J) : T(A) \to F(J) \}_{j \in J} \right)$$

Bibliography

[1] Samson Abramsky and Achim Jung. *Domain theory*, page 1–168. Oxford University Press, Inc., USA, 1995. ISBN 019853762X.

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353. ACM, 2009. doi: 10.1145/1480881.1480925. URL https://doi.org/10.1145/1480881.1480925.

[3] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015. doi: 10.1017/S0960129514000486. URL https://doi.org/10.1017/S0960129514000486.

[4] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 195–207. ACM, 2017. doi: 10.1145/3018610.3018613. URL https://doi.org/10.1145/3018610.3018613.

[5] Pierre America and Jan J. M. M. Rutten. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989. doi: 10.1016/0022-0000(89)90027-5. URL https://doi.org/10.1016/0022-0000(89)90027-5.

[6] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi: 10.1145/504709.504712. URL https://doi.org/10.1145/504709.504712.

[7] Kenichi Asai. Logical relations for call-by-value delimited continuations. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth*

Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005, volume 6 of *Trends in Functional Programming*, pages 63–78. Intellect, 2005.

[8] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic Delimited Continuations. In Zhong Shao, editor, *Programming Languages and Systems*, pages 239–254. Springer, 2007. ISBN 978-3-540-76637-7. doi: 10.1007/978-3-540-76637-7_16.

[9] Robert Atkey. Relational Parametricity for Higher Kinds. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 46–61. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi: 10.4230/LIPIcs.CSL.2012.46. URL https://doi.org/10.4230/LIPIcs.CSL.2012.46.

[10] Lennart Augustsson and Kent Petersson. Silly Type Families. Available at https://web.cecs.pdx.edu/~sheard/papers/silly.pdf, September 1994.

[11] Steve Awodey. *Category Theory*. Oxford Logic Guides. Oxford University Press, London, England, 2 edition, June 2010. ISBN 9780199237180.

[12] Casper Bach Poulsen and Cas van der Rest. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proceedings of the ACM on Programming Languages*, 7(POPL):62:1801–62:1831, January 2023. doi: 10.1145/3571255.

[13] H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993. ISBN 0198537611.

[14] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1): 108–123, January 2015. ISSN 2352-2208. doi: 10.1016/j.jlamp.2014.02.001.

[15] Nick Benton, Andrew Kennedy, and Carsten Varming. Some Domain Theory and Denotational Semantics in Coq. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2009. doi: 10.1007/978-3-642-03359-9_10. URL https://doi.org/10.1007/978-3-642-03359-9_10.

[16] Ulrich Berger and Helmut Schwichtenberg. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 203–211. IEEE Computer Society, 1991. doi:

10.1109/LICS.1991.151645. URL https://doi.org/10.1109/LICS.1991.151645.

[17] Malgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. In António Porto and Francisco Javier López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 289–300. ACM, 2009. doi: 10.1145/1599410.1599446. URL https://doi.org/10.1145/1599410.1599446.

[18] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Log. Methods Comput. Sci.*, 1(2), 2005. doi: 10.2168/LMCS-1(2:5)2005. URL https://doi.org/10.2168/LMCS-1(2:5)2005.

[19] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290319. URL https://doi.org/10.1145/3290319.

[20] Richard S. Bird and Lambert G. L. T. Meertens. Nested Datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998. doi: 10.1007/BFb0054285. URL https://doi.org/10.1007/BFb0054285.

[21] Lars Birkedal and Aleš Bizjak. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic, 2017. URL http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.

[22] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theor. Comput. Sci.*, 411(47): 4102–4122, 2010. doi: 10.1016/J.TCS.2010.07.010. URL https://doi.org/10.1016/j.tcs.2010.07.010.

[23] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 119–132. ACM, 2011. doi: 10.1145/1926385.1926401. URL https://doi.org/10.1145/1926385.1926401.

[24] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.*, 8(4), 2012. doi: 10.2168/LMCS-8(4:1)2012. URL https://doi.org/10.2168/LMCS-8(4:1)2012.

[25] Lars Birkedal, Ales Bizjak, and Jan Schwinghammer. Step-Indexed Relational Reasoning for Countable Nondeterminism. *Log. Methods Comput. Sci.*, 9(4), 2013. doi: 10.2168/LMCS-9(4:4)2013. URL https://doi.org/10.2168/LMCS-9(4:4)2013.

[26] Errett Bishop. *Foundations of Constructive Analysis*. Mcgraw-Hill, New York, NY, USA, 1967.

[27] Ales Bizjak, Lars Birkedal, and Marino Miculan. A Model of Countable Nondeterminism in Guarded Type Theory. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2014. doi: 10.1007/978-3-319-08918-8_8. URL https://doi.org/10.1007/978-3-319-08918-8_8.

[28] Robert Cartwright and Matthias Felleisen. Extensible Denotational Language Specifications. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 244–272, Berlin, Heidelberg, 1994. Springer. ISBN 978-3-540-48383-0. doi: 10.1007/3-540-57887-0_99.

[29] James Cheney and Ralf Hinze. First-Class Phantom Types. Technical report, Cornell University, 2003.

[30] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and Reasoning with Guarded Recursion for Coinductive Types. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2015. doi: 10.1007/978-3-662-46678-0_26. URL https://doi.org/10.1007/978-3-662-46678-0_26.

[31] Olivier Danvy. Type-Directed Partial Evaluation. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 242–257. ACM Press, 1996. doi: 10.1145/237721.237784. URL https://doi.org/10.1145/237721.237784.

[32] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical report, DIKU – Computer Science Department, University of Copenhagen, 1989.

[33] Olivier Danvy and Andrzej Filinski. Abstracting Control. In Gilles Kahn, editor, *Proceedings of the 1990 ACM Conference on LISP and Functional*

Programming, LFP 1990, Nice, France, 27-29 June 1990, pages 151–160. ACM, 1990. doi: 10.1145/91556.91622. URL https://doi.org/10.1145/91556.91622.

[34] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434314. URL https://doi.org/10.1145/3434314.

[35] Robert Dockins. Formalized, Effective Domain Theory in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014. doi: 10.1007/978-3-319-08970-6_14. URL https://doi.org/10.1007/978-3-319-08970-6_14.

[36] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.*, 7(2), 2011. doi: 10.2168/LMCS-7(2:16)2011. URL https://doi.org/10.2168/LMCS-7(2:16)2011.

[37] Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing First-Class Continuations in ML. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 163–173. ACM Press, 1991. doi: 10.1145/99583.99608. URL https://doi.org/10.1145/99583.99608.

[38] Jana Dunfield and Neelakantan R. Krishnaswami. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290322. URL https://doi.org/10.1145/3290322.

[39] Peter Dybjer. Inductive families. *Formal Aspects Comput.*, 6(4):440–465, 1994. doi: 10.1007/BF01211308. URL https://doi.org/10.1007/BF01211308.

[40] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming*, 17(6):687–730, 2007. ISSN 1469-7653, 0956-7968. doi: 10.1017/S0956796807006259. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/monadic-framework-for-delimited-continuations/D99D1394370DFA8EA8428D552B5D8E7E.

[41] Matthias Felleisen. Reflections on landins's j-operator: A partly historical note. *Comput. Lang.*, 12(3/4):197–207, 1987. doi: 10.1016/0096-0551(87)90022-1. URL https://doi.org/10.1016/0096-0551(87)90022-1.

[42] Matthias Felleisen. The Theory and Practice of First-Class Prompts. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190. ACM Press, 1988. doi: 10.1145/73560.73576. URL https://doi.org/10.1145/73560.73576.

[43] Matthias Felleisen. On the Expressive Power of Programming Languages. *Sci. Comput. Program.*, 17(1-3):35–75, 1991. doi: 10.1016/0167-6423(91)90036-W. URL https://doi.org/10.1016/0167-6423(91)90036-W.

[44] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, pages 193–222. North-Holland, 1987.

[45] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full jumps. In Jérôme Chailloux, editor, *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP 1988, Snowbird, Utah, USA, July 25-27, 1988*, pages 52–62. ACM, 1988. doi: 10.1145/62678.62684. URL https://doi.org/10.1145/62678.62684.

[46] Marcelo P. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming, October 6-8, 2002, Pittsburgh, PA, USA (Affiliated with PLI 2002)*, pages 26–37. ACM, 2002. doi: 10.1145/571157.571161. URL https://doi.org/10.1145/571157.571161.

[47] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 193–202. IEEE Computer Society, 1999. doi: 10.1109/LICS.1999.782615. URL https://doi.org/10.1109/LICS.1999.782615.

[48] Peter Freyd. Algebraically complete categories. In Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini, editors, *Category Theory*, pages 95–104, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-46435-8.

[49] Dan Frumin, Amin Timany, and Lars Birkedal. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.*, 8(POPL):332–361, 2024. doi: 10.1145/3632854. URL https://doi.org/10.1145/3632854.

[50] Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with gadts. In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013. doi: 10.1007/978-3-319-03542-0_19. URL https://doi.org/10.1007/978-3-319-03542-0_19.

[51] Pietro Di Gianantonio and Marino Miculan. A Unifying Approach to Recursive and Co-recursive Definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 2002. doi: 10.1007/3-540-39185-1_9. URL https://doi.org/10.1007/3-540-39185-1_9.

[52] Pietro Di Gianantonio and Marino Miculan. Unifying Recursive and Co-recursive Definitions in Sheaf Categories. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2987 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2004. doi: 10.1007/978-3-540-24727-2_11. URL https://doi.org/10.1007/978-3-540-24727-2_11.

[53] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.*, 3(POPL): 3:1–3:28, 2019. doi: 10.1145/3290316. URL https://doi.org/10.1145/3290316.

[54] Jean-Yves Girard. A Fixpoint Theorem in Linear Logic, February 1992. Post to Linear Logic mailing list, http://www.seas.upenn.edu/~sweirich/types/archive/1992/msg00030.html.

[55] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2014. doi: 10.1007/978-3-319-08970-6_18. URL https://doi.org/10.1007/978-3-319-08970-6_18.

[56] Carl A. Gunter. Universal Profinite Domains. *Inf. Comput.*, 72(1):1–30, 1987. doi: 10.1016/0890-5401(87)90048-4. URL https://doi.org/10.1016/0890-5401(87)90048-4.

[57] Ryu Hasegawa. Categorical Data Types in Parametric Polymorphism. *Math. Struct. Comput. Sci.*, 4(1):71–109, 1994. doi: 10.1017/S0960129500000372. URL https://doi.org/10.1017/S0960129500000372.

[58] Michael Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *J. Funct. Program.*, 8(4):413–436, 1998. doi: 10.1017/S0956796898003153. URL https://doi.org/10.1017/s0956796898003153.

[59] Jason Z. S. Hu and Jacques Carette. Formalizing category theory in Agda. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 327–342. ACM, 2021. doi: 10.1145/3437992.3439922. URL https://doi.org/10.1145/3437992.3439922.

[60] Gérard P. Huet and Amokrane Saïbi. Constructive category theory. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 239–276. The MIT Press, 2000.

[61] Brian Huffman. A Purely Definitional Universal Domain. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2009. doi: 10.1007/978-3-642-03359-9_19. URL https://doi.org/10.1007/978-3-642-03359-9_19.

[62] Iris team. The Iris Project website and Coq development, 2025. URL https://iris-project.org/.

[63] Jr. James Hiram Morris. *Lambda-calculus models of programming languages*. Phd thesis, Massachusetts Institute of Technology, Cambridge, MA, 1968. MAC-TR-57, available at https://dspace.mit.edu/handle/1721.1/64850.

[64] Thomas Jech. *Set Theory*. Springer Berlin Heidelberg, 2003. ISBN 9783540440857. doi: 10.1007/3-540-44761-x. URL http://dx.doi.org/10.1007/3-540-44761-X.

[65] Patricia Johann and Pierre Cagne. How Functorial Are (Deep) GADTs? *CoRR*, abs/2203.14891, 2022. doi: 10.48550/arXiv.2203.14891. URL https://doi.org/10.48550/arXiv.2203.14891.

[66] Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*,

pages 297–308. ACM, 2008. doi: 10.1145/1328438.1328475. URL https://doi.org/10.1145/1328438.1328475.

[67] Patricia Johann and Enrico Ghiorzi. Parametricity for Nested Types and GADTs. *Log. Methods Comput. Sci.*, 17(4), 2021. doi: 10.46298/lmcs-17(4:23)2021. URL https://doi.org/10.46298/lmcs-17(4:23)2021.

[68] Patricia Johann and Enrico Ghiorzi. (Deep) induction rules for GADTs. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 324–337. ACM, 2022. doi: 10.1145/3497775.3503680. URL https://doi.org/10.1145/3497775.3503680.

[69] Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. Parametricity for Primitive Nested Types. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 2021. doi: 10.1007/978-3-030-71995-1_17. URL https://doi.org/10.1007/978-3-030-71995-1_17.

[70] Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. GADTs, Functoriality, Parametricity: Pick Two. In Mauricio Ayala-Rincón and Eduardo Bonelli, editors, *Proceedings 16th Logical and Semantic Frameworks with Applications, LSFA 2021, Buenos Aires, Argentina (Online), 23rd - 24th July, 2021*, volume 357 of *EPTCS*, pages 77–92, 2021. doi: 10.4204/EPTCS.357.6. URL https://doi.org/10.4204/EPTCS.357.6.

[71] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In John H. Reppy and Julia Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61. ACM, 2006. doi: 10.1145/1159803.1159811. URL https://doi.org/10.1145/1159803.1159811.

[72] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016. doi: 10.1145/2951913.2951943. URL https://doi.org/10.1145/2951913.2951943.

[73] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:

e20, 2018. doi: 10.1017/S0956796818000151. URL https://doi.org/10.1017/S0956796818000151.

[74] Max Kelly. *Basic concepts of enriched category theory series number 64*. London Mathematical Society lecture note series. Cambridge University Press, Cambridge, England, February 1982. ISBN 9780521287029.

[75] Alexis King. Delimited continuation primops. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst, 2021. Accessed: 2024-06-27.

[76] Dominik Kirst and Gert Smolka. Large model constructions for second-order ZF in dependent type theory. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 228–239. ACM, 2018. doi: 10.1145/3167095. URL https://doi.org/10.1145/3167095.

[77] Oleg Kiselyov and Chung-chieh Shan. A Substructural Type System for Delimited Continuations. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 223–239, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-73228-0. doi: 10.1007/978-3-540-73228-0_17.

[78] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 234–248, New York, NY, USA, January 2019. Association for Computing Machinery. ISBN 978-1-4503-6222-1. doi: 10.1145/3293880.3294106.

[79] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi: 10.1145/3009837.3009855. URL https://doi.org/10.1145/3009837.3009855.

[80] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018. doi: 10.1145/3236772. URL https://doi.org/10.1145/3236772.

[81] Peter J. Landin. A correspondence between ALGOL 60 and church's lambda-notations: Part II. *Commun. ACM*, 8(3):158–167, 1965. doi: 10.1145/363791.363804. URL https://doi.org/10.1145/363791.363804.

[82] Peter J. Landin. A generalization of jumps and labels. *High. Order Symb. Comput.*, 11(2):125–143, 1998. doi: 10.1023/A:1010068630801. URL https://doi.org/10.1023/A:1010068630801.

[83] Saunders Mac Lane. *Categories for the working mathematician*. Graduate Texts in Mathematics. Springer, New York, NY, 2 edition, September 1998. ISBN 978-0-387-98403-2.

[84] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014. doi: 10.4204/EPTCS.153.8. URL https://doi.org/10.4204/EPTCS.153.8.

[85] Daan Leijen. Structured asynchrony with algebraic effects. In Sam Lindley and Brent A. Yorgey, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, pages 16–29. ACM, 2017. doi: 10.1145/3122975.3122977. URL https://doi.org/10.1145/3122975.3122977.

[86] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi: 10.1016/S0890-5401(03)00088-9. URL https://doi.org/10.1016/S0890-5401(03)00088-9.

[87] Saunders MacLane and Ieke Moerdijk. *Sheaves in geometry and logic*. Universitext. Springer, New York, NY, 1 edition, May 1992. ISBN 978-0-387-97710-2.

[88] Marek Materzok and Dariusz Biernacki. Subtyping Delimited Continuations. *ACM SIGPLAN Notices*, 46(9):81–93, 2011. ISSN 0362-1340. doi: 10.1145/2034574.2034786. URL https://doi.org/10.1145/2034574.2034786.

[89] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

[90] Hiroshi Nakano. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266. IEEE Computer Society, 2000. doi: 10.1109/LICS.2000.855774. URL https://doi.org/10.1109/LICS.2000.855774.

[91] Christine Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi*

and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. doi: 10.1007/BFb0037116. URL https://doi.org/10.1007/BFb0037116.

[92] Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From Semantics to Computer Science*, pages 383–413. Cambridge University Press, 2009. URL https://inria.hal.science/inria-00431806.

[93] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. A Model of PCF in Guarded Type Theory. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 333–349. Elsevier, 2015. doi: 10.1016/J.ENTCS.2015.12.020. URL https://doi.org/10.1016/j.entcs.2015.12.020.

[94] Daniel Peebles, James Deikun, Ulf Norell, Dan Doel, Darius Jahandarie, and James Cook. categories: Categories parametrized by morphism equality in Agda. https://github.com/copumpkin/categories, 2018.

[95] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, Microsoft Research, July 2004. URL https://www.microsoft.com/en-us/research/publication/wobbly-types-type-inference-for-generalised-algebraic-data-types/. Microsoft Research.

[96] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

[97] Andrew M. Pitts. Typed Operational Reasoning. In Benjamin C. Pierce, editor, *Advanced Topics In Types And Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2004. ISBN 0262162288.

[98] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977. doi: 10.1016/0304-3975(77)90044-5. URL https://doi.org/10.1016/0304-3975(77)90044-5.

[99] Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:3–15, 2004. doi: 10.1016/J.JLAP.2004.03.009. URL https://doi.org/10.1016/j.jlap.2004.03.009.

[100] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[101] Gordon D. Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, December 2013. ISSN 1860-5974. doi: 10.2168/LMCS-9(4:23)2013.

[102] Piotr Polesiuk. IxFree: Step-indexed logical relations in Coq. In *3 rd International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[103] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 232–244. ACM, 2006. doi: 10.1145/1111037.1111058. URL https://doi.org/10.1145/1111037.1111058.

[104] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972. doi: 10.1145/800194.805852. URL https://doi.org/10.1145/800194.805852.

[105] Edmund P. Robinson and Giuseppe Rosolini. Reflexive Graphs and Parametric Polymorphism. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 364–371. IEEE Computer Society, 1994. doi: 10.1109/LICS.1994.316053. URL https://doi.org/10.1109/LICS.1994.316053.

[106] Peter Schroeder-Heister. Definitional reflection and the completion. In Roy Dyckhoff, editor, *Extensions of Logic Programming*, pages 333–347, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48417-2.

[107] Dana Scott. OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION. Technical Report PRG02, OUCL, November 1970.

[108] Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136, Berlin, Heidelberg, 1972. Springer Berlin Heidelberg. ISBN 978-3-540-37609-5.

[109] Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

[110] Tim Sheard. Putting curry-howard to work. In Daan Leijen, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, pages 74–85. ACM, 2005. doi: 10.1145/1088348.1088356. URL https://doi.org/10.1145/1088348.1088356.

[111] Tim Sheard and Emir Pasalic. Meta-programming With Built-in Type Equality. *Electron. Notes Theor. Comput. Sci.*, 199:49–65, 2008. doi: 10.1016/j.entcs.2007.11.012. URL https://doi.org/10.1016/j.entcs.2007.11.012.

[112] Filip Sieczkowski, Ales Bizjak, and Lars Birkedal. ModuRes: A Coq Library for Modular Reasoning About Concurrent Higher-Order Imperative Programming Languages. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2015. doi: 10.1007/978-3-319-22102-1_25. URL https://doi.org/10.1007/978-3-319-22102-1_25.

[113] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. The essence of generalized algebraic data types, October 2023. URL https://doi.org/10.5281/zenodo.10040534.

[114] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. The essence of generalized algebraic data types. *Proc. ACM Program. Lang.*, 8 (POPL):695–723, 2024. doi: 10.1145/3632866. URL https://doi.org/10.1145/3632866.

[115] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for Noninterference with Interaction Trees. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 29:1–29:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPICS.ECOOP.2023.29. URL https://doi.org/10.4230/LIPIcs.ECOOP.2023.29.

[116] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP Symb. Comput.*, 3(1):67–99, 1990.

[117] Dorai Sitaram and Matthias Felleisen. Models of Continuations without Continuations. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 185–196. Association for Computing Machinery, 1991. ISBN 978-0-89791-419-2. doi: 10.1145/99583.99611. URL https://dl.acm.org/doi/10.1145/99583.99611.

[118] Michael B. Smyth and Gordon D. Plotkin. The Category-Theoretic Solution of Recursive Domain Equations. *SIAM J. Comput.*, 11(4):761–783, 1982. doi: 10.1137/0211062. URL https://doi.org/10.1137/0211062.

[119] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada,*

June 20-25, 2021, pages 80–95. ACM, 2021. doi: 10.1145/3453483.3454031. URL https://doi.org/10.1145/3453483.3454031.

[120] Sergei Stepanenko and Amin Timany. Solving guarded domain equations in presheaves over ordinals and mechanizing it. In Maribel Fernández, editor, *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, July 14-20, 2025, Birmingham, UK*, volume 337 of *LIPIcs*, pages 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi: 10.4230/LIPICS.FSCD.2025.33. URL https://doi.org/10.4230/LIPIcs.FSCD.2025.33.

[121] Sergei Stepanenko and Amin Timany. The Rocq Mechanization of Solving Guarded Domain Equations in Presheaves Over Ordinals. https://doi.org/10.5281/zenodo.15406039, 2025.

[122] Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, and Lars Birkedal. Context-dependent effects in guarded interaction trees. In Viktor Vafeiadis, editor, *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*, volume 15695 of *Lecture Notes in Computer Science*, pages 286–313. Springer, 2025. doi: 10.1007/978-3-031-91121-7_12. URL https://doi.org/10.1007/978-3-031-91121-7_12.

[123] Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, and Lars Birkedal. Context-Dependent Effects in Guarded Interaction Trees, January 2025. URL https://doi.org/10.5281/zenodo.14623650.

[124] Jonathan Sterling and Robert Harper. Guarded Computational Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 879–888, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5583-4. doi: 10.1145/3209108.3209153. URL http://doi.acm.org/10.1145/3209108.3209153.

[125] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[126] Christopher S. Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *High. Order Symb. Comput.*, 13(1/2):135–152, 2000. doi: 10.1023/A:1010026413531. URL https://doi.org/10.1023/A:1010026413531.

[127] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN*

International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007, pages 53–66. ACM, 2007. doi: 10.1145/1190315.1190324. URL https://doi.org/10.1145/1190315.1190324.

[128] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput.*, 11(4):405–439, December 1998. ISSN 1388-3690. doi: 10.1023/A:1010035624696. URL https://doi.org/10.1023/A:1010035624696.

[129] Amin Timany. Commuting Quantifiers, Oct 2020. URL https://cs.au.dk/~timany/blog/commuting_quantifiers/. Blog post, accessed on Feb 8, 2025.

[130] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP): 105:1–105:28, 2019. doi: 10.1145/3341709. URL https://doi.org/10.1145/3341709.

[131] Amin Timany and Bart Jacobs. Category Theory in Coq 8.5. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-010-1. doi: 10.4230/LIPIcs.FSCD.2016.30. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2016.30.

[132] Amin Timany, Armaël Guéneau, and Lars Birkedal. The Logical Essence of Well-Bracketed Control Flow. *Proc. ACM Program. Lang.*, 8(POPL):575–603, 2024. doi: 10.1145/3632862. URL https://doi.org/10.1145/3632862.

[133] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A Logical Approach to Type Soundness. *Journal of the ACM*, 71, 2024.

[134] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[135] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent Effects for Reusable Language Components. In Hakjoo Oh, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 182–201, Cham, 2021. Springer International Publishing. ISBN 978-3-030-89051-3. doi: 10.1007/978-3-030-89051-3_11.

[136] Max Vistrup, Michael Sammler, and Ralf Jung. Program logics à la carte. *Proc. ACM Program. Lang.*, 9(POPL):300–331, 2025. doi: 10.1145/3704847. URL https://doi.org/10.1145/3704847.

[137] Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(2):175–210, 2010. doi: 10.1017/S0956796810000079. URL https://doi.org/10.1017/S0956796810000079.

[138] Philip Wadler. Theorems for Free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404. URL https://doi.org/10.1145/99370.99404.

[139] Mitchell Wand. Fixed-Point Constructions in Order-Enriched Categories. *Theor. Comput. Sci.*, 8:13–30, 1979. doi: 10.1016/0304-3975(79)90053-7. URL https://doi.org/10.1016/0304-3975(79)90053-7.

[140] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 275–286. ACM, 2013. doi: 10.1145/2500365.2500599. URL https://doi.org/10.1145/2500365.2500599.

[141] John Wiegley. category-theory: Category Theory in Coq. https://github.com/jwiegley/category-theory, 2019.

[142] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi: 10.1006/inco.1994.1093. URL https://doi.org/10.1006/inco.1994.1093.

[143] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA, September 2014. Association for Computing Machinery. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633358.

[144] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi: 10.1145/604131.604150. URL https://doi.org/10.1145/604131.604150.

[145] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119.

[146] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proceedings of the ACM on Programming Languages*, 5(ICFP): 67:1–67:30, August 2021. doi: 10.1145/3473572.

[147] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-188-7. doi: 10.4230/LIPIcs.ITP.2021.32.