University of California

Los Angeles

Typed Self-Applicable Meta-Programming

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Matthew Scott Brown

2017

Abstract of the Dissertation

Typed Self-Applicable Meta-Programming

by

Matthew Scott Brown
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2017
Professor Jens Palsberg, Chair

Self-applicable meta-programming has its roots in the early days of computer science. Two early examples were published in 1936: the universal Turing machine, and a self-interpreter for the λ-calculus. These were major advances in computability theory, but self-application has a long history of practical uses as well. Many languages have self-interpreters or self-hosting compilers. Others support self-applicable meta-programming as a general-purpose technique that enables elegant solutions to many problems. Until now, these techniques have been incompatible with static type checking, which has many benefits of its own. In this thesis I present techniques for practical self-applicable meta-programming for statically typed languages.

The dissertation of Matthew Scott Brown is approved.

Miryung Kim

Todd Millstein

Yiannis Moschovakis

Jens Palsberg, Committee Chair

University of California, Los Angeles

2017

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, I thank my advisor Jens Palsberg for his valuable advice, guidance, and encouragement, for many enlightening discussions, and for his unflagging support throughout my meandering explorations.

I am grateful to Todd Millstein for his advice and encouragement, and for his many comments that helped to clarify the ideas in this thesis. I am also grateful for the friendship of many people from the UCLA Computer Science department, especially those from the Compilers lab and the Programming Languages and Software Engineering reading group.

Finally, I especially thank my wife Carrie and my family, for their love, support and patience.

Vita

2011 - 2013	Master of Science, Computer Science University of California, Los Angeles Advisor: Jens Palsberg Thesis: Typed Self-Optimization
May 2012	Second Summer School on Formal Techniques Menlo College, Atherton, CA
2001 - 2005	Bachelor of Science, Computer Science University of California, Santa Cruz

Publications

Brown, M., and Palsberg, J. Self-representation in Girard's System U. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL 2015, ACM, pp. 471–484.

Brown, M., and Palsberg, J. Breaking through the normalization barrier: A self-interpreter for F-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL 2016, ACM, pp. 5–17.

Brown, M., and Palsberg, J. Typed self-evaluation via intensional type functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 2017), POPL 2017, ACM, pp. 415–428.

CHAPTER 1

Introduction

Self-applicable meta-programming has its roots in the early days of computer science. Two early examples were published in 1936: the universal Turing machine [87], and a self-interpreter for the λ-calculus [57]. A universal Turing machine can simulate any Turing machine given a representation of that machine. In particular, it can simulate itself. A self-interpreter for λ-calculus is similar: it can simulate any λ-term by interpreting its representation. In particular, it can interpret its own representation. These were major advances in the theory of computability and uncomputability, but self-application has a long history of practical benefit as well.

In 1960, McCarthy published his famous Lisp self-interpreter [59]. Since then, self-interpreters have been implemented for many other popular languages, including Haskell [66], JavaScript [39], Python [73], Ruby [95], Scheme [5], and Standard ML [74]. A self-interpreter enables the language designer to easily modify, extend, and grow the language, and implement tools like debuggers and IDEs [71, 18]. Other well-known examples of self-applicable meta-programs include self-hosting compilers and virtual machines.

Static type checking is a fundamental technique in the field of programming languages. It makes software more reliable by eliminating a large class of errors and enforcing abstraction and modularity. Compilers can use static type information in a variety of ways to optimize programs. Static types can also help programmers understand their code, as a kind of checked documentation that cannot go stale, and can enable powerful tools for programmers.

Self-applicable meta-programming promises the same benefits for statically typed languages that they already provide for dynamically typed ones. More than that, type checking meta-programs themselves can make meta-programming safer, more efficient, and less error-prone. There are several techniques for type checking meta-programs and program representations. One option is to use a universal type like `String` or `Exp` for all representations. This has been used to define the self-interpreters for Haskell [66] and Standard ML [74], both statically typed languages. There are several limitations to a universally typed representation. First, the type system cannot guarantee that a representation represents a well-typed term. As a result, an interpreter can introduce a type error into its input program, and the error will go undetected by the type checker. Second, some meta-programs are not supported by the universal-type approach. An important example is a self-recognizer, a kind of self-interpreter that recovers a term from its representation. The result of the self-recognizer is well-typed only if its input represents a well-typed term. Since the type checker can't ensure a representation encodes a well typed term, it can't type check the self-recognizer itself.

Another approach, called *typed representation*, ensures that a representation encodes a well-typed program of a particular type. Here, the type of a representation is parameterized by the type of the term it represents. For example, if a term has a type `T`, then its representation might have the type `Exp T`. Using a typed representation ensures that the result of the self-recognizer will be well-typed. Due to the correspondence between the types of programs and the types of their representations, a typed representation scheme can support a self-recognizer. For example, a typed self-recognizer might have the type $\forall$`T. Exp T` $\to$ `T`. The polymorphic type allows the self-recognizer to interpret any well-typed representation. Another kind of self-interpreter, called a self-evaluator, maps the representation of a term to a representation of its value. A self-evaluator for typed representations might have the type $\forall$`T. Exp T` $\to$ `Exp T`. This type ensures that the evaluator always outputs a well-typed representation, and that the input and output programs have the same type. Thus, typed representation can guarantee strong correctness properties of meta-programs.

The challenge of combining self-application and typed representation was set forth in 1991 by Pfenning and Lee [69]. They studied whether System F could support a typed self-interpreter, and concluded that it "seems to be impossible". No further progress was made until 2009, when Rendel, Ostermann and Hofer showed that a statically typed language can support a typed self-recognizer [71]. In 2011, Jay and Palsberg presented a statically typed combinator calculus that supports a typed self-recognizer and a typed self-evaluator [52]. These seminal works demonstrated that typed self-application can be achieved.

The thesis of this dissertation is that self-applicable meta-programming is feasible and useful for statically typed programming languages. My results show that the combination of static type checking and self-applicable meta-programming provides greater benefits than simply the sum of its parts. It not only enables for statically-typed languages the meta-programming techniques that have long been a staple of dynamically typed languages; type checking meta-programs themselves can ensure strong correctness guarantess that would otherwise be difficult to ensure. The techniques developed herein include innovations in language design, program representation, and meta-programming. To maximize practicality, we consider only languages that are type-safe, that support decidable type checking, and that include only well-known language constructs. All novelty with respect to language design is in the variations and combination of these language constructs. Finally, our typed self-representation schemes are designed to be general – expressive enough to program a variety of useful meta-programs, including self-recognizers, self-evaluators, and more.

Chapter 2 presents a typed self-recognizer for System F_ω, a well-known language with decidable typechecking. F_ω has been called “the workhorse of modern compilers” [65] due to its foundational role in many typed functional programming languages. It is also *strongly normalizing*, which informally means that all programs always terminate. That a self-interpreter is even possible for a strongly normalizing language like F_ω contradicted the prior conventional wisdom, and is one of the most surprising results of this dissertation.

Are all self-interpreters for a single language equal? Are they equally useful? Are they equally difficult to implement? Can any language that supports a self-recognizer also support a self-evaluator? These are questions we pick up in Chapter 3, which focuses on typed self-evaluation. We present a language $F_\omega^{\mu i}$ that supports both kinds of self-interpreter. This is the second main result of this dissertation. $F_\omega \mu i$ extends F_ω with support for recursive functions and a typed self-representation encoded as a Generalized Algebraic Data Type (GADT). The results of Chapter 3 indicate that self-evaluation requires a more powerful language than self-recognition; it remains an open question whether F_ω – or any other strongly normalizing language – can support self-evaluation. In exchange, self-evaluation provides more power than self-recognition alone: a self-evaluator can implement a particular evaluation strategy, while a self-recognizer inherits the strategy from the meta-level.

The control over evaluation enabled by self-evaluation leads to an important practical application: a typed self-applicable partial evaluator for $\mathrm{F}_\omega^{\mu i}$. This is the third main result of this dissertation, and is the focus of Chapter 4. A partial evaluator is related to Kleene's s-m-n theorem. Given a program that takes $m + n$ inputs, of which m are known statically, a partial evaluator produces a version of the program specialized to the first m inputs. When given the remaining n inputs, the specialized program will run more efficiently than the original program. Partial evaluation has applications in compilation, optimization and generation of programs [53]. Further, a *self-applicable* partial evaluator can be combined with a self-interpreter to automatically generate compilers and compiler generators [41].

Despite the long history of self-applicable partial evaluation, achieving it for a statically typed language has remained an open problem. Chapter 4 solves this problem by presenting the first typed self-applicable partial evaluator, specifically for $\mathrm{F}_\omega^{\mu i}$. It can be used to automatically generate a compiler and compiler generator via the Futamura projections [41]. Further it is Jones-optimal, meaning that specialization can remove all the overhead of a self-interpreter.

Chapter 2 is based on the paper "Breaking through the Normalization Barrier: A Self-interpreter for F-omega" [20], and chapter 3 is based on the paper "Typed Self-Evaluation via Intensional Type Functions"[21]. Proofs of the theorems stated in those chapters are included in the full version of those papers, which are available from their web pages [1, 2]. Those web pages also provide the software artifacts for the papers, which include implementations of the languages and meta-programs, as well as many tests and examples.

CHAPTER 2

Breaking through the Normalization Barrier: A Self-Interpreter for F_ω

2.1 Introduction

Barendregt's notion of a *self-interpreter* is a program that recovers a program from its representation and is implemented in the language itself [10]. Specifically for λ-calculus, the challenge is to devise a quoter that maps each term $\mathtt{e}$ to a representation $\overline{\mathtt{e}}$, and a self-interpreter $\mathtt{u}$ (for *unquote*) such that for every λ-term $\mathtt{e}$ we have $(\mathtt{u}\ \overline{\mathtt{e}}) \equiv_\beta \mathtt{e}$. The quoter is an injective function from λ-terms to representations, which are λ-terms in normal form. Barendregt used Church numerals as representations, while in general one can use any β-normal terms as representations. For untyped λ-calculus, in 1936 Kleene presented the first self-interpreter [57], and in 1992 Mogensen presented the first *strong* self-interpreter $\mathtt{u}$ that satisfies the property $(\mathtt{u}\ \overline{\mathtt{e}}) \longrightarrow_\beta^* \mathtt{e}$ [63]. In 2009, Rendel, Ostermann, and Hofer [71] presented the first self-interpreter for a *typed* λ-calculus (F_ω^*), and in previous work [19] we presented the first self-interpreter for a typed λ-calculus with *decidable* type checking (Girard's System U). Those results are all for non-normalizing λ-calculi and they go about as far as one can go before reaching what we call the *normalization barrier*.

The normalization barrier: According to conventional wisdom, a self-interpreter for a strongly normalizing λ-calculus is impossible.

The normalization barrier stems from a theorem in computability theory that says that a total universal function for the total computable functions is impossible. Several books, papers, and web pages have concluded that the theorem about total universal functions carries

$$\begin{tikzcd} F \arrow[loop above] \arrow[r, "{[69]}"] & F_\omega \arrow[loop above] \arrow[r, "{[69]}"] & F^+_\omega \arrow[loop above] \arrow[r, "{[19]}", "|" marking] & U \arrow[loop above, "{[19]}"] \end{tikzcd}$$

normalization barrier

Figure 2.1: Four typed λ-calculi: $\rightarrow$ denotes "represented in."

over to self-interpreters for strongly normalizing languages. For example, Turner states that "For any language in which all programs terminate, there are always terminating programs which cannot be written in it - among these are the interpreter for the language itself" [88, pg. 766]. Similarly, Stuart writes that "Total programming languages are still very powerful and capable of expressing many useful computations, but one thing they can't do is interpret themselves" [80, pg. 264]. Additionally, the Wikipedia page on the *Normalization Property* (accessed in May 2015) explains that a self-interpreter for a strongly normalizing λ-calculus is impossible. That Wikipedia page cites three typed λ-calculi, namely simply typed λ-calculus, System F, and the Calculus of Constructions, each of which is a member of Barendregt's cube of typed λ-calculi [11]. We can easily add examples to that list, particularly the other five corners of Barendregt's λ-cube, including F_ω. The normalization barrier implies that a self-interpreter is impossible for every language in the list. In a seminal paper in 1991 Pfenning and Lee [69] considered whether one can define a self-interpreter for System F or F_ω and found that the answer seemed to be "no".

In this chapter we take up the challenge presented by the normalization barrier.

The challenge: Can we define a self-interpreter for a strongly normalizing λ-calculus?

Our result: Yes, we present a strong self-interpreter for the strongly normalizing λ-calculus F_ω; the program representation is *deep* and supports a variety of other operations. We also present a much simpler self-interpreter that works for each of System F, F_ω, and F^+_ω; the program representation is *shallow* and supports no other operations.

Figure 2.1 illustrates how our result relates to other representations of typed λ-calculi with decidable type checking. The normalization barrier separates the three strongly-normalizing languages on the left from System U on the right, which is not strongly-normalizing. Pfenning and Lee represented System F in F_ω, and F_ω in F_ω^+. In previous work we showed that F_ω^+ can be represented in System U, and that System U can represent itself. This chapter contributes the unlabeled self-loops on F, F_ω, and F_ω^+, depicting the first self-representations for strongly-normalizing languages.

Our result breaks through the normalization barrier. The conventional wisdom underlying the normalization barrier makes an implicit assumption that all representations will behave like their counterpart in the computability theorem, and therefore the theorem must apply to them as well. This assumption excludes other notions of representation, about which the theorem says nothing. Thus, our result does not contradict the theorem, but shows that the theorem is less far-reaching than previously thought.

Our result relies on three technical insights. First, we observe that the proof of the classical theorem in computability theory relies on a diagonalization gadget, and that a typed representation can ensure that the gadget fails to type check in F_ω, so the proof doesn't necessarily carry over to F_ω. Second, for our deep representation we use a novel *extensional* approach to representing polymorphic terms. We use instantiation functions that describe the relationship between a quantified type and one of its instance types. Each instantiation function takes as input a term of a quantified type, and instantiates it with a particular parameter type. Third, for our deep representation we use a novel representation of types, which helps us type check a continuation-passing-style transformation.

We present five self-applicable operations on our deep representation, namely a strong self-interpreter, a continuation-passing-style transformation, an intensional predicate for testing whether a closed term is an abstraction or an application, a size measure, and a normal-form checker. Our list of operations extends those of previous work [19].

Our deep self-representation of F_ω could be useful for type-checking self-applicable meta-programs, with potential for applications in typed macro systems, partial evaluators, compilers, and theorem provers. In particular, F_ω is a subset of the proof language of the Coq proof assistant, and Morrisett has called F_ω *the workhorse of modern compilers* [65].

Our deep representation is the most powerful self-representation of F_ω that we have identified: it supports all the five operations listed above. One can define several other representations for F_ω by using fewer of our insights. Ultimately, one can define a *shallow* representation that supports only a self-interpreter and nothing else. As a stepping stone towards explaining our main result, we will show a shallow representation and a self-interpreter in Section 3.3. That representation and self-interpreter have the distinction of working for System F, F_ω and F^+_ω. Thus, we have solved the two challenges left open by Pfenning and Lee [69].

Rest of the chapter. In Section 2 we describe F_ω, in Section 3 we analyze the normalization barrier, in Section 4 we describe instantiation functions, in Section 5 we show how to represent types, in Section 6 we show how to represent terms, in Section 7 we present our operations on program representations, in Section 8 we discuss our implementation and experiments, in Section 9 we discuss various aspects of our result, and in Section 10 we compare with related work.

2.2 System F_ω

System F_ω is a typed λ-calculus within the λ-cube [11]. It combines two axes of the cube: polymorphism and higher-order types (type-level functions). In this section we summarize the key properties of System F_ω used in this chapter. We refer readers interested in a complete tutorial to other sources [11, 70]. We give a definition of F_ω in Figure 2.2. It includes a grammar, rules for type formation and equivalence, and rules for term formation and reduction. The grammar defines the kinds, types, terms, and environments. As usual, types classify terms, kinds classify types, and environments classify free term and type variables. Every syntactically well-formed kind and environment is legal, so we do not include separate formation rules for them. The type formation rules determine the legal types in a given environment, and assigns a kind to each legal type. Similarly, the term formation rules determine the legal terms in a given environment, and assigns a type to each legal term.

$$\begin{array}{rl}
\text{(kinds)} & \kappa ::= * \mid \kappa_1 \to \kappa_2 \\
\text{(types)} & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
\text{(terms)} & \mathsf{e} ::= \mathsf{x} \mid \lambda\mathsf{x}{:}\tau.\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \Lambda\alpha{:}\kappa.\mathsf{e} \mid \mathsf{e}\ \tau \\
\text{(environments)} & \Gamma ::= \langle\rangle \mid \Gamma,(\mathsf{x}{:}\tau) \mid \Gamma,(\alpha{:}\kappa)
\end{array}$$

Grammar

$$\frac{(\alpha{:}\kappa) \in \Gamma}{\Gamma \vdash \alpha : \kappa}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma,(\alpha{:}\kappa) \vdash \tau : *}{\Gamma \vdash (\forall\alpha{:}\kappa.\tau) : *}$$

$$\frac{\Gamma,(\alpha{:}\kappa_1) \vdash \tau : \kappa_2}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau) : \kappa_1 \to \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}$$

Type Formation

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau \equiv \sigma}{\sigma \equiv \tau} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_1 \equiv \sigma_1 \qquad \tau_2 \equiv \sigma_2}{\tau_1 \to \tau_2 \equiv \sigma_1 \to \sigma_2} \qquad \frac{\tau \equiv \sigma}{(\forall\alpha{:}\kappa.\tau) \equiv (\forall\alpha{:}\kappa.\sigma)}$$

$$\frac{\tau \equiv \sigma}{(\lambda\alpha{:}\kappa.\tau) \equiv (\lambda\alpha{:}\kappa.\sigma)} \qquad \frac{\tau_1 \equiv \sigma_1 \qquad \tau_2 \equiv \sigma_2}{\tau_1\ \tau_2 \equiv \sigma_1\ \sigma_2}$$

$$\frac{}{(\lambda\alpha{:}\kappa.\tau) \equiv (\lambda\beta{:}\kappa.\tau[\alpha := \beta])} \qquad \frac{}{(\lambda\alpha{:}\kappa.\tau)\ \sigma \equiv (\tau[\alpha := \sigma])}$$

Type Equivalence

$$\frac{(\mathsf{x}{:}\tau) \in \Gamma}{\Gamma \vdash \mathsf{x} : \tau}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma,(\mathsf{x}{:}\tau_1) \vdash \mathsf{e} : \tau_2}{\Gamma \vdash (\lambda\mathsf{x}{:}\tau_1.\mathsf{e}) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \tau_2 \to \tau \qquad \Gamma \vdash \mathsf{e}_2 : \tau_2}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \tau}$$

$$\frac{\Gamma,(\alpha{:}\kappa) \vdash \mathsf{e} : \tau}{\Gamma \vdash (\Lambda\alpha{:}\kappa.\mathsf{e}) : (\forall\alpha{:}\kappa.\tau)}$$

$$\frac{\Gamma \vdash \mathsf{e} : (\forall\alpha{:}\kappa.\tau) \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash \mathsf{e}\ \sigma : \tau[\alpha{:=}\sigma]}$$

$$\frac{\Gamma \vdash \mathsf{e} : \tau \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathsf{e} : \sigma}$$

Term Formation

$$(\lambda\mathsf{x}{:}\tau.\mathsf{e})\ \mathsf{e}_1 \longrightarrow \mathsf{e}[\mathsf{x} := \mathsf{e}_1]$$
$$(\Lambda\alpha{:}\kappa.\mathsf{e})\ \tau \longrightarrow \mathsf{e}[\alpha := \tau]$$

$$\frac{\mathsf{e}_1 \longrightarrow \mathsf{e}_2}{\begin{array}{c}\mathsf{e}_1\ \mathsf{e}_3 \longrightarrow \mathsf{e}_2\ \mathsf{e}_3 \\ \mathsf{e}_3\ \mathsf{e}_1 \longrightarrow \mathsf{e}_3\ \mathsf{e}_2 \\ \mathsf{e}_1\ \tau \longrightarrow \mathsf{e}_2\ \tau \\ (\lambda\mathsf{x}{:}\tau.\mathsf{e}_1) \longrightarrow (\lambda\mathsf{x}{:}\tau.\mathsf{e}_2) \\ (\Lambda\alpha{:}\kappa.\mathsf{e}_1) \longrightarrow (\Lambda\alpha{:}\kappa.\mathsf{e}_2)\end{array}}$$

Reduction

Figure 2.2: Definition of System F_ω

Our definition is similar to Pierce's [70], with two differences: we use a slightly different syntax, and our semantics is arbitrary β-reduction instead of call-by-value.

It is well known that type checking is decidable, and that types of F_ω terms are unique up to equivalence. We will write $\mathsf{e} \in F_\omega$ to mean "e is a well-typed term in F_ω". Any well-typed term in System F_ω is strongly normalizing, meaning there is no infinite sequence of β-reductions starting from that term. If we β-reduce enough times, we will eventually reach a term in β-normal form that cannot be reduced further. Formally, term e is β-normal if there is no e' such that $\mathsf{e} \longrightarrow \mathsf{e}'$. We require that representations of terms be *data*, which for λ-calculus usually means a term in β-normal form.

2.3 The Normalization Barrier

In this section, we explore the similarity of a universal computable function in computability theory and a self-interpreter for a programming language. As we shall see, the exploration has a scary beginning and a happy ending. At first, a classical theorem in computability theory seems to imply that a self-interpreter for F_ω is impossible. Fortunately, further analysis reveals that the proof relies on an assumption that a diagonalization gadget can always be defined for a language with a self-interpreter. We show this assumption to be false: by using a *typed representation*, it is possible to define a self-interpreter such that the diagonalization gadget cannot possibly type check. We conclude the section by demonstrating a simple typed self-representation and a self-interpreter for F_ω.

2.3.1 Functions from Numbers to Numbers

We recall a classical theorem in computability theory (Theorem 2.3.2). The proof of the theorem is a diagonalization argument, which we divide into two steps: first we prove a key property (Theorem 2.3.1) and then we proceed with the proof of Theorem 2.3.2.

Let $\mathbb{N}$denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Let $\bar{\cdot}$ be an injective function that maps each total, computable function in $\mathbb{N} \to \mathbb{N}$ to an element of $\mathbb{N}$.

We say that $\mathsf{u} \in (\mathbb{N} \times \mathbb{N}) \rightharpoonup \mathbb{N}$ is a universal function for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, if for every total, computable function f in $\mathbb{N} \to \mathbb{N}$, we have $\forall \mathsf{v} \in \mathbb{N}$: $\mathtt{u(\overline{f},v) = f(v)}$. The symbol $\rightharpoonup$ denotes that u may be a partial function. Indeed, Theorem 2.3.2 proves that u *must* be partial. We let $\mathsf{U}niv(\mathbb{N} \to \mathbb{N})$ denote the set of universal functions for the total, computable functions in $\mathbb{N} \to \mathbb{N}$.

Given a function u in $(\mathbb{N} \times \mathbb{N}) \rightharpoonup \mathbb{N}$, we define the function $\mathsf{p_u}$ in $\mathbb{N} \to \mathbb{N}$, where $\mathtt{p_u(x) = u(x,x) + 1}$.

Theorem 2.3.1. *If* $\mathsf{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$*, then* $\mathsf{p_u}$ *isn't total.*

Proof. Suppose $\mathsf{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$ and $\mathsf{p_u}$ is total. Notice that $\mathsf{p_u}$ is a total, computable function in $\mathbb{N} \to \mathbb{N}$ so $\overline{\mathsf{p_u}}$ is defined. We calculate:

$$\mathtt{p_u(\overline{p_u}) = u(\overline{p_u},\overline{p_u}) + 1 = p_u(\overline{p_u}) + 1}$$

Given that $\mathtt{p_u}$ is total, we have that $\mathtt{p_u(\overline{p_u})}$ is defined; let us call the result v. From $\mathtt{p_u(\overline{p_u}) = p_u(\overline{p_u}) + 1}$, we get $\mathtt{v = v + 1}$, which is impossible. So we have reached a contradiction, hence our assumption (that $\mathtt{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$ and $\mathtt{p_u}$ is total) is wrong. We conclude that if $\mathtt{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$, then $\mathtt{p_u}$ isn't total. □

Theorem 2.3.2. *If* $\mathtt{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$*, then* $\mathtt{u}$ *isn't total.*

Proof. Suppose $\mathtt{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$ and $\mathtt{u}$ is total. For every $\mathtt{x} \in \mathbb{N}$, we have that $\mathtt{p_u(x) = u(x,x) + 1}$. Since $\mathtt{u}$ is total, $\mathtt{u(x,x) + 1}$ is defined, and therefore $\mathtt{p_u(x)}$ is also defined. Since $\mathtt{p_u(x)}$ is defined for every $\mathtt{x} \in \mathbb{N}$, $\mathtt{p_u}$ is total. However, Theorem 2.3.1 states that $\mathtt{p_u}$ is not total. Thus we have reached a contradiction, so our assumption (that $\mathtt{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$ and $\mathtt{u}$ is total) is wrong. We conclude that if $\mathtt{u} \in \mathsf{U}niv(\mathbb{N} \to \mathbb{N})$, then $\mathtt{u}$ isn't total. □

Intuitively, Theorem 2.3.2 says that if we write an interpreter for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, then that interpreter must go into an infinite loop on some inputs.

2.3.2 Typed λ-Calculus: $\mathbf{F}_\omega$

Does Theorem 2.3.2 imply that a self-interpreter for F_ω is impossible? Recall that every well-typed term in F_ω is strongly normalizing. So, if we have a self-interpreter $\mathtt{u}$ for F_ω and we have $\mathtt{(u\ e)} \in \mathrm{F}_\omega$, then $\mathtt{(u\ e)}$ is strongly normalizing, which is intuitively expresses that $\mathtt{u}$ is a total function. Thus, Theorem 2.3.2 seems to imply that a self-interpreter for F_ω is impossible. This is the normalization barrier. Let us examine this intuition via a "translation" of Section 2.3.1 to F_ω.

Let us recall the definition of a self-interpreter from Section 1, here for F_ω. A quoter is an injective function from terms in F_ω to their representations, which are β-normal terms in F_ω. We write $\overline{\mathtt{e}}$ to denote the representation of a term $\mathtt{e}$. We say that $\mathtt{u} \in \mathrm{F}_\omega$ is a self-interpreter for F_ω, if $\forall \mathtt{e} \in \mathrm{F}_\omega$: $\mathtt{(u\ \overline{e})} \equiv_\beta \mathtt{e}$. We allow $\mathtt{(u\ \overline{e})}$ to include type abstractions or applications as necessary, and leave them implicit. We use $\mathsf{S}elfInt(\mathrm{F}_\omega)$ to denote the set of self-interpreters for F_ω.

Notice a subtle difference between the definition of a universal function in Section 2.3.1 and the definition of a self-interpreter. The difference is that a universal function takes both its arguments at the same time, while, intuitively, a self-interpreter is *curried* and takes its arguments one by one. This difference plays no role in our further analysis.

Notice also the following consequences of the two requirements of a quoter. The requirement that a quoter must produce terms in β-normal form rules out the identity function as a quoter, because it maps reducible terms to reducible terms. The requirement that a quoter must be injective rules out the function that maps each term to its normal form, because it maps β-equivalent terms to the same β-normal form.

The proof of Theorem 2.3.1 relies on the diagonalization gadget $(\mathtt{p_u}\ \overline{\mathtt{p_u}})$, where $\mathtt{p_u}$ is a cleverly defined function. The idea of the proof is to achieve the equality $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) = (\mathtt{p_u}\ \overline{\mathtt{p_u}}) + 1$. For the F_ω version of Theorem 2.3.1, our idea is to achieve the equality $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \equiv_\beta \lambda\mathtt{y}.(\mathtt{p_u}\ \overline{\mathtt{p_u}})$, where $\mathtt{y}$ is fresh. Here, $\lambda\mathtt{y}$ plays the role of "+1". Given $\mathtt{u} \in \mathrm{F}_\omega$, we define $\mathtt{p_u} = \lambda\mathtt{x}.\ \lambda\mathtt{y}.\ ((\mathtt{u}\ \mathtt{x})\ \mathtt{x})$, where x,y are fresh, and where we omit suitable type annotations for x,y. We can now state an F_ω version of Theorem 2.3.1.

Theorem 2.3.3. *If* $\mathtt{u} \in SelfInt(\mathrm{F}_\omega)$*, then* $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \notin F_\omega$*.*

Proof. Suppose $\mathtt{u} \in \mathsf{S}elfInt(\mathrm{F}_\omega)$ and $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \in \mathrm{F}_\omega$. We calculate:

$$\begin{aligned} & \mathtt{p_u}\ \overline{\mathtt{p_u}} \\ \equiv_\beta\ & \lambda\mathtt{y}.\ ((\mathtt{u}\ \overline{\mathtt{p_u}})\ \overline{\mathtt{p_u}}) \\ \equiv_\beta\ & \lambda\mathtt{y}.\ (\mathtt{p_u}\ \overline{\mathtt{p_u}}) \end{aligned}$$

From $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \in \mathrm{F}_\omega$ we have that $(\mathtt{p_u}\ \overline{\mathtt{p_u}})$ is strongly normalizing. From the Church-Rosser property of F_ω, we have that $(\mathtt{p_u}\ \overline{\mathtt{p_u}})$ has a unique normal form; let us call it v. From $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \equiv_\beta \lambda\mathtt{y}.(\mathtt{p_u}\ \overline{\mathtt{p_u}})$ we get $\mathtt{v} \equiv_\beta \lambda\mathtt{y}.\mathtt{v}$. Notice that $\mathtt{v}$ and $\lambda\mathtt{y}.\mathtt{v}$ are *distinct* yet β-equivalent normal forms. Now the Church-Rosser property implies that β-equivalent terms must have the *same* normal form. Thus $\mathtt{v} \equiv_\beta \lambda\mathtt{y}.\mathtt{v}$ implies $\mathtt{v} \equiv_\alpha \lambda\mathtt{y}.\mathtt{v}$, which is false. So we have reached a contradiction, hence our assumption (that $\mathtt{u} \in \mathsf{S}elfInt(\mathrm{F}_\omega)$ and $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \in \mathrm{F}_\omega$) is wrong. We conclude that if $\mathtt{u} \in \mathsf{S}elfInt(\mathrm{F}_\omega)$, then $(\mathtt{p_u}\ \overline{\mathtt{p_u}}) \notin \mathrm{F}_\omega$. □

What is an F_ω version of Theorem 2.3.2? Given that every term in F_ω is "total" in the sense described earlier, Theorem 2.3.2 suggests that we should expect $\mathsf{S}elfInt(F_\omega) = \emptyset$. However this turns out to be wrong and indeed in this chapter we will define a self-representation and self-interpreter for F_ω. So, $\mathsf{S}elfInt(F_\omega) \neq \emptyset$.

We saw earlier that Theorem 2.3.1 helped prove Theorem 2.3.2. Why does Theorem 2.3.3 fail to lead the conclusion $\mathsf{S}elfInt(F_\omega) = \emptyset$? Observe that in the proof of Theorem 2.3.2, the key step was to notice that if $\mathtt{u}$ is total, also $\mathtt{p}_\mathtt{u}$ is total, which contradicts Theorem 2.3.1. In contrast, the assumption $\mathtt{u} \in \mathsf{S}elfInt(F_\omega)$ produces no useful conclusion like $(\mathtt{p}_\mathtt{u}\ \overline{\mathtt{p}_\mathtt{u}}) \in F_\omega$ that would contradict Theorem 2.3.3. In particular, it is possible for $\mathtt{u}$ and $\mathtt{p}_\mathtt{u}$ to be typeable in F_ω, and yet for $(\mathtt{p}_\mathtt{u}\ \overline{\mathtt{p}_\mathtt{u}})$ to be untypeable. So, the door is open for a self-interpreter for F_ω.

2.3.3 A Self-Interpreter for F_ω

Inspired by the optimism that emerged in Section 2.3.2, let us now define a quoter and a self-interpreter for F_ω. The quoter will support *only* the self-interpreter and nothing else. The idea of the quoter is to use a designated variable `id` to block the reduction of every application. The self-interpreter unblocks reduction by substituting the polymorphic identity function for `id`. Below we define the representation $\overline{\mathtt{e}}$ of a closed term `e`.

$$\Gamma \vdash \mathtt{x} : \tau \rhd \mathtt{x}$$

$$\frac{\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e} : \tau_2 \rhd \mathtt{q}}{\Gamma \vdash (\lambda\mathtt{x}{:}\tau_1.\mathtt{e}) : \tau_1 \to \tau_2 \rhd (\lambda\mathtt{x}{:}\tau_1.\mathtt{q})}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau \rhd \mathtt{id}\ (\tau_2 \to \tau)\ \mathtt{q}_1\ \mathtt{q}_2}$$

$$\frac{\Gamma,\alpha{:}\kappa \vdash \mathtt{e} : \tau \rhd \mathtt{q}}{\Gamma \vdash (\Lambda\alpha{:}\kappa.\mathtt{e}) : (\forall\alpha{:}\kappa.\tau) \rhd (\Lambda\alpha{:}\kappa.\mathtt{q})}$$

$$\frac{\Gamma \vdash \mathtt{e} : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathtt{q} \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \mathtt{e}\ \tau_2 : (\tau_1[\alpha := \tau_2]) \rhd \mathtt{id}\ (\forall\alpha{:}\kappa.\tau_1)\ \mathtt{q}\ \tau_2}$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q} \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathtt{e} : \sigma \rhd \mathtt{q}}$$

$$\frac{\langle\rangle \vdash \texttt{e} : \tau \rhd \texttt{q}}{\overline{\texttt{e}} = \lambda\texttt{id}{:}(\forall\alpha{:}{*}.\ \alpha \to \alpha).\ \texttt{q}}$$

Our representation is defined in two steps. First, the rules of the form $\Gamma \vdash \texttt{e} : \tau \rhd \texttt{q}$ build a pre-representation `q` from the typing judgment of a term `e`. The types are needed to instantiate each occurrence of the designated variable `id`. The representation $\overline{\texttt{e}}$ is defined by abstracting over `id` in the pre-representation. Our self-interpreter takes a representation as input and applies it to the polymorphic identity function:

$$\begin{aligned}\texttt{unquote} &: \forall\alpha{:}{*}.\ ((\forall\beta{:}{*}.\beta \to \beta) \to \alpha) \to \alpha \\ &= \Lambda\alpha{:}{*}.\ \lambda\texttt{q}{:}(\forall\beta{:}{*}.\beta \to \beta) \to \alpha. \\ &\quad\ \texttt{q}\ (\Lambda\beta{:}{*}.\ \lambda\texttt{x}{:}\beta.\ \texttt{x})\end{aligned}$$

Theorem 2.3.4.
If $\langle\rangle \vdash \texttt{e} : \tau$, then $\langle\rangle \vdash \overline{\texttt{e}} : (\forall\alpha{:}{*}.\ \alpha \to \alpha) \to \tau$ and $\overline{\texttt{e}}$ is in β-normal form.

Theorem 2.3.5.
If $\langle\rangle \vdash \texttt{e} : \tau$, then $\texttt{unquote}\ \tau\ \overline{\texttt{e}} \longrightarrow^{*} \texttt{e}$.

This self-interpreter demonstrates that it is possible to break through the normalization barrier. In fact, we can define a similar self-representation and self-interpreter for System F and for System F_ω^+. However, the representation supports no other operations than `unquote`: parametricity implies that the polymorphic identity function is the *only* possible argument to a representation $\overline{\texttt{e}}$ [91]. The situation is similar to the one faced by Pfenning and Lee who observed that “evaluation is just about the only useful function definable” for their representation of F_ω in F_ω^+. We call a representation *shallow* if it supports only one operation, and we call representation *deep* if it supports a variety of operations. While the representation above is shallow, we have found it to be a good starting point for designing deep representations.

In Figure 2.3 we define a *deep* self-representation of F_ω that supports multiple operations, including a self-interpreter, a CPS-transformation, and a normal-form checker. The keys to why this works are two novel techniques along with typed Higher-Order Abstract Syntax (HOAS), all of which we will explain in the following sections. First, in Section 2.4 we present an extensional approach to representing polymorphism in F_ω. Second, in Section 2.5 we present a simple representation of types that is sufficient to support our CPS transformation. Third, in Section 2.6 we present a typed HOAS representation based on Church encoding, which supports operations that fold over the term. Finally, in Section 2.7 we define five operations for our representation.

2.4 Representing Polymorphism

In this section, we discuss our extensional approach to representing polymorphic terms in our type Higher-Order Abstract Syntax representation. Our approach allows us to define our HOAS representation of F_ω in F_ω itself. Before presenting our extensional approach, we will review the intensional approach used by previous work. As a running example, we consider how to program an important piece of a self-interpreter for a HOAS representation.

Our HOAS representation, like those of Pfenning and Lee [69], Rendel et al. [71], and our own previous work [19], is based on a typed Church encoding. Operations are defined by cases functions, one for each of λ-abstraction, application, type abstraction, and type application. Our representation differs from the previous work in how we type check the case functions for type abstraction and type applications. Our running example will focus just on the case function for type applications. To simplify further, we consider only the case function for type applications in a self-interpreter.

2.4.1 The Intensional Approach

The approach of previous work [69, 71, 19] can be demonstrated by a *polymorphic type-application function*, which can apply any polymorphic term to any type in its domain. The function `tapp`$^+$ defined below is a polymorphic type application function for System F_ω^+. System F_ω^+ extends F_ω with kind abstractions and applications in terms (written $\Lambda\kappa$.`e` and `e` κ respectively), and kind-polymorphic types (written $\forall^+\kappa.\tau$):

$$\texttt{tapp}^+ : (\forall^+\kappa.\forall\beta{:}\kappa\to*.(\forall\alpha{:}\kappa.\ \beta\ \alpha) \to \forall\gamma{:}\kappa.\beta\ \gamma)$$
$$\texttt{tapp}^+ = \Lambda^+\kappa.\Lambda\beta{:}\kappa\to*.\lambda\texttt{e}{:}(\forall\alpha{:}\kappa.\beta\ \alpha).\ \Lambda\gamma{:}\kappa.\texttt{e}\ \gamma$$

The variables κ and β respectively range over the domain and codomain of an arbitrary quantified type. The domain of $(\forall\alpha_1{:}\kappa_1.\tau_1)$ is the kind κ_1, and the codomain is the type function $(\lambda\alpha_1{:}\kappa_1.\tau_1)$ since τ_1 depends on a type parameter α_1 of kind κ_1. Since the body of a quantified type must have kind $*$, the codomain function $(\lambda\alpha_1{:}\kappa_1.\ \tau_1)$ must have kind $(\kappa_1 \to *)$. A quantified type can be expressed in terms of its domain and codomain: $(\forall\alpha_1{:}\kappa_1.\tau_1) \equiv (\forall\alpha{:}\kappa_1.\ (\lambda\alpha_1{:}\kappa_1.\tau_1)\ \alpha)$. Similarly, any instance of the quantified type can be expressed as an application of the codomain to a parameter: $(\tau_1[\alpha_1{:=}\tau_2]) \equiv (\lambda\alpha_1{:}\kappa_1.\tau_1)\ \tau_2$. We use these equivalences in the type of $\mathtt{tapp}^+$: the quantified type $(\forall\alpha{:}\kappa.\ \beta\ \alpha)$ is expressed in terms of an arbitrary domain κ and codomain β, and the instantiation $\beta\ \gamma$ is expressed as an application of the codomain β to an arbitrary parameter γ.

We call this encoding *intensional* because it encodes the structure of a quantified type by abstracting over its parts (its domain κ and codomain β). This ensures that e can only have a quantified type, and that γ ranges over exactly the types to which e can be applied. In other words, γ can be instantiated with τ_2 if and only if e τ_2 is well-typed.

Consider a type application e τ_2 with the derivation:

$$\frac{\Gamma \vdash \mathtt{e} : (\forall\alpha_1{:}\kappa_1.\tau_1) \qquad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \mathtt{e}\ \tau_2 : \tau_1[\alpha_1{:=}\tau_2]}$$

We can encode e τ_2 in F_ω^+ as $\mathtt{tapp}^+\kappa_1\ (\lambda\alpha_1{:}\kappa_1.\tau_1)\ \mathtt{e}\ \tau_2$. However, F_ω does not support kind polymorphism, so $\mathtt{tapp}^+$ is not definable in F_ω. To represent F_ω in itself, we need a new approach.

2.4.2 An Extensional Approach

The approach we use in this chapter is *extensional:* rather than encoding the structure of a quantified type, we encode the relationship between a quantified type and its instances. We encode the relationship "$(\tau_1[\alpha{:=}\tau_2])$ is an instance of $(\forall\alpha_1{:}\kappa_1.\tau_1)$" with an *instantiation function* of type $(\forall\alpha_1{:}\kappa_1.\tau_1) \to (\tau_1[\alpha_1{:=}\tau_2])$. An example of such an instantiation function is $\lambda\mathtt{x}{:}(\forall\alpha_1{:}\kappa_1.\tau_1).\ \mathtt{x}\ \tau_2$ that instantiates an input term of type $(\forall\alpha_1{:}\kappa_1.\tau_1)$ with the type τ_2. For convenience, we define an abbreviation $\mathtt{inst}_{(\tau,\sigma)} = \lambda\mathtt{x}{:}\tau.\ \mathtt{x}\ \sigma$, which is well-typed only when τ is a quantified type and σ is in the domain of τ.

The advantage of using instantiation functions is that all quantified types and all instantiations of quantified types are types of kind $*$. Thus, we can encode the rule for type applications in F_ω by abstracting over the quantified type, the instance type, and the instantiation function for them:

```
tapp : (∀α:*. α → ∀β:*. (α → β) → β)
tapp = Λα:*. λe:α. Λβ:*. λinst:α → β. inst e
```

Using `tapp` we can encode the type application `e` τ_2 above as

$$\texttt{tapp}\ (\forall\alpha_1{:}\kappa_1.\tau_1)\ \texttt{e}\ (\tau_1[\alpha_1{:=}\tau_2])\ \texttt{inst}_{((\forall\alpha1:\kappa1.\tau1),\tau2)}$$

Unlike the intensional approach, the extensional approach provides no guarantee that `e` will always have a quantified type. Furthermore, even if `e` does have a quantified type, `inst` is not guaranteed to actually be an instantiation function. In short, the intensional approach provides two Free Theorems [91] that we don't get with our extensional approach. However, the extensional approach has the key advantage of enabling a representation of F_ω in itself.

2.5 Representing Types

We use type representations to type check term representations and operations on term representations. Our type representation is shown as part of the representation of F_ω in Figure 2.3. The $[\![\tau]\!]$ syntax denotes the pre-representation of the type τ, and $\overline{\tau}$ denotes the representation. A pre-representation is defined using a designated variable `F`, and a representation abstracts over `F`.

Our type representation is novel and designed to support three important properties: first, it can represent all types (not just types of kind $*$); second, representation preserves equivalence between types; third, it is expressive enough to typecheck all our benchmark operations. The first and second properties and play an important part in our representation of polymorphic terms.

$$
\begin{aligned}
\llbracket\alpha\rrbracket &= \alpha \\
\llbracket\tau_1 \to \tau_2\rrbracket &= \mathtt{F}\ \llbracket\tau_1\rrbracket \to \mathtt{F}\ \llbracket\tau_2\rrbracket \\
\llbracket\forall\alpha{:}\kappa.\tau\rrbracket &= \forall\alpha{:}\kappa.\ \mathtt{F}\ \llbracket\tau\rrbracket \\
\llbracket\lambda\alpha{:}\kappa.\tau\rrbracket &= \lambda\alpha{:}\kappa.\ \llbracket\tau\rrbracket \\
\llbracket\tau_1\ \tau_2\rrbracket &= \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket
\end{aligned}
$$

Pre-Representation of Types

$$\overline{\tau} = \lambda\mathtt{F}{:}{*}\to{*}.\ \llbracket\tau\rrbracket$$

Representation of Types

$$
\begin{array}{l}
\mathtt{U}\ \ = (* \to *) \to * \\
\mathtt{Op} = \lambda\mathtt{F}{:}* \to *.\ \lambda\alpha{:}\mathtt{U}.\ \mathtt{F}\ (\alpha\ \mathtt{F}) \\[1ex]
\mathtt{Strip} = \lambda\mathtt{F}{:}* \to *.\ \lambda\alpha{:}*. \\
\qquad \forall\beta{:}*.(\forall\gamma{:}*.\ \mathtt{F}\ \gamma \to \beta) \to \alpha \to \beta \\[1ex]
\mathtt{Abs}\ \ = \lambda\mathtt{F}{:}*\to*. \\
\qquad \forall\alpha{:}*.\forall\beta{:}*.(\mathtt{F}\ \alpha \to \mathtt{F}\ \beta) \to \mathtt{F}\ (\mathtt{F}\ \alpha \to \mathtt{F}\ \beta) \\
\mathtt{App}\ \ = \lambda\mathtt{F}{:}*\to*. \\
\qquad \forall\alpha{:}*.\forall\beta{:}*.\mathtt{F}\ (\mathtt{F}\ \alpha \to \mathtt{F}\ \beta) \to \mathtt{F}\ \alpha \to \mathtt{F}\ \beta \\
\mathtt{TAbs} = \lambda\mathtt{F}{:}*\to*. \\
\qquad \forall\alpha{:}*.\ \mathtt{Strip}\ \mathtt{F}\ \alpha \to \alpha \to \mathtt{F}\ \alpha \\
\mathtt{TApp} = \lambda\mathtt{F}{:}*\to*. \\
\qquad \forall\alpha{:}*.\mathtt{F}\ \alpha \to \forall\beta{:}*.(\alpha \to \mathtt{F}\ \beta) \to \mathtt{F}\ \beta \\[1ex]
\mathtt{Exp} = \lambda\alpha{:}\mathtt{U}.\ \forall\mathtt{F}{:}* \to *. \\
\qquad \mathtt{Abs}\ \mathtt{F} \to \mathtt{App}\ \mathtt{F} \to \mathtt{TAbs}\ \mathtt{F} \to \mathtt{TApp}\ \mathtt{F} \to \\
\qquad \mathtt{Op}\ \mathtt{F}\ \alpha
\end{array}
$$

Kind and Type Definitions

$$\mathtt{inst}_{(\tau,\sigma)} = \lambda\mathtt{x}{:}\tau.\ \mathtt{x}\ \sigma$$

Instantiation Functions

$$\frac{}{* \blacktriangleright (\forall\alpha{:}*.\alpha)} \qquad \frac{\kappa \blacktriangleright \sigma}{\kappa_1 \to \kappa \blacktriangleright \lambda\alpha{:}\kappa_1.\sigma}$$

Kind Inhabitants

$$\frac{\kappa \blacktriangleright \sigma}{\begin{array}{l}\mathtt{strip}_{(\mathtt{F},\kappa,\tau)} = \Lambda\alpha{:}*.\ \lambda\mathtt{f}{:}(\forall\beta{:}*.\ \mathtt{F}\,\beta \to \alpha). \\ \qquad \lambda\mathtt{x}{:}(\forall\gamma{:}\kappa.\mathtt{F}\,(\tau\,\gamma)).\ \mathtt{f}\,(\tau\,\sigma)\,(\mathtt{x}\,\sigma)\end{array}}$$

Strip Functions

$$\frac{(\mathtt{x}{:}\tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau \rhd \mathtt{x}}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e} : \tau_2 \rhd \mathtt{q}}{\Gamma \vdash (\lambda\mathtt{x}{:}\tau_1.\mathtt{e}) : \tau_1 \to \tau_2 \rhd \mathtt{abs}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ (\lambda\mathtt{x}{:}\mathtt{F}\ \llbracket\tau_1\rrbracket.\mathtt{q})}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau \rhd \mathtt{app}\ \llbracket\tau_2\rrbracket\ \llbracket\tau\rrbracket\ \mathtt{q}_1\ \mathtt{q}_2}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \mathtt{e} : \tau \rhd \mathtt{q}}{\begin{array}{l}\Gamma \vdash (\Lambda\alpha{:}\kappa.\mathtt{e}) : (\forall\alpha{:}\kappa.\tau) \rhd \mathtt{tabs}\ \llbracket\forall\alpha{:}\kappa.\tau\rrbracket \\ \qquad\qquad\qquad \mathtt{strip}_{(\mathtt{F},\kappa,\llbracket\lambda\alpha:\kappa.\tau\rrbracket)} \\ \qquad\qquad\qquad (\Lambda\alpha{:}\kappa.\mathtt{q})\end{array}}$$

$$\frac{\Gamma \vdash \mathtt{e} : (\forall\alpha{:}\kappa.\tau) \rhd \mathtt{q} \qquad \Gamma \vdash \sigma : \kappa}{\begin{array}{l}\Gamma \vdash \mathtt{e}\,\sigma : \tau[\alpha{:=}\sigma] \rhd \mathtt{tapp}\ \llbracket\forall\alpha{:}\kappa.\tau\rrbracket\ \mathtt{q}\ \llbracket\tau[\alpha{:=}\sigma]\rrbracket \\ \qquad\qquad\qquad \mathtt{inst}_{(\llbracket\forall\alpha:\kappa.\tau\rrbracket,\llbracket\sigma\rrbracket)}\end{array}}$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q} \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathtt{e} : \sigma \rhd \mathtt{q}}$$

Pre-Representation of Terms

$$\frac{\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}}{\begin{array}{l}\overline{\mathtt{e}} = \Lambda\mathtt{F}{:}* \to *. \\ \qquad \lambda\mathtt{abs}{:}\mathtt{Abs}\,\mathtt{F}.\ \ \lambda\mathtt{app}{:}\mathtt{App}\,\mathtt{F}. \\ \qquad \lambda\mathtt{tabs}{:}\mathtt{TAbs}\,\mathtt{F}.\ \lambda\mathtt{tapp}{:}\mathtt{TApp}\,\mathtt{F}. \\ \qquad \mathtt{q}\end{array}}$$

Representation of Terms

Figure 2.3: Representation of F_ω

Type representations support operations that iterate a type function `R` over the first-order types – arrows and universal quantifiers. Each operation on representations produces results of the form `R (`$[\![\tau]\!]$`[F := R])`, which we call the "interpretation of τ under `R`". For example, the interpretation of $(\forall\alpha{:}{*}.\ \alpha \to \alpha)$ under `R` is `R (`$[\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]$`[F := R])` $=$ `R (`$\forall\alpha{:}{*}.$ `R (R` $\alpha \to$ `R` α`))`.

As stated previously, type representations are used to typecheck representations of terms and their operations. In particular, a term of type τ is represented by a term of type `Exp` $\overline{\tau}$, and each operation on term representation produces results with types that are interpretations under some `R`.

Let's consider the outputs produced by `unquote`, `size`, and `cps`, when applied to a representation of the polymorphic identity function, which has the type $(\forall\alpha{:}{*}.\ \alpha \to \alpha)$. For `unquote`, the type function `R` is the identity function `Id = (`$\lambda\alpha{:}{*}.\alpha$`)`. Therefore, `unquote` applied to the representation of the polymorphic identity function will produce an output with the type `Id (`$\forall\alpha{:}{*}.$ `Id (Id` $\alpha \to$ `Id` α`))` $\equiv$ $(\forall\alpha{:}{*}.\alpha \to \alpha)$. For `size`, `R` is the constant function `KNat = (`$\lambda\alpha$`:*.Nat)`. Therefore, `size` applied to the representation of the polymorphic identity function will produce an output with the type `KNat (`$\forall\alpha{:}{*}.$ `KNat (KNat` $\alpha \to$ `KNat` α`))` $\equiv$ `Nat`. For `cps`, `R` is the function `Ct = (`$\lambda\alpha{:}{*}.\ \forall\beta{:}{*}.\ (\alpha \to \beta) \to \beta$`)`, such that `Ct` α is the type of a continuation for values of type α. Therefore, `cps` applied to the representation of the polymorphic identity function will produce an output with the type `Ct (`$\forall\alpha{:}{*}.$ `Ct (Ct` $\alpha \to$ `Ct` α`))`. This type suggests that every sub-term has been transformed into continuation-passing style.

We also represent higher-order types, since arrows and quantifiers can occur within them. Type variables, abstractions, and applications are represented meta-circularly. Intuitively, the pre-representation of an abstraction is an abstraction over pre-representations . Since pre-representations of κ-types (i.e. types of kind κ) are themselves κ-types, an abstraction over κ-types can also abstract over pre-representations of κ-types. In other words, abstractions are represented as themselves. The story is the same for type variables and applications.

Examples. The representation of $(\forall\alpha{:}*.\ \alpha \to \alpha)$ is:

$$\overline{\forall\alpha{:}*.\ \alpha \to \alpha}$$
$$= \lambda\mathsf{F}{:}*{\to}*.\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]$$
$$= \lambda\mathsf{F}{:}*{\to}*.\ \forall\alpha{:}*.\ \mathsf{F}\ (\mathsf{F}\ \alpha \to \mathsf{F}\ \alpha)$$

Our representation is defined so that the representations of two β-equivalent types are also β-equivalent. In other words, representation of types preserves β-equivalence. In particular, we can normalize a type before or after representation, with the same result. For example,

$$\begin{aligned}
& \overline{\forall\alpha{:}*.(\lambda\gamma{:}*.\gamma \to \gamma)\ \alpha} \\
= \ & \lambda\mathsf{F}{:}*{\to}*.\ [\![\forall\alpha{:}*.(\lambda\gamma{:}*.\gamma \to \gamma)\ \alpha]\!] \\
= \ & \lambda\mathsf{F}{:}*{\to}*.\ \forall\alpha{:}*.\ \mathsf{F}\ ((\lambda\gamma{:}*.\mathsf{F}\ \gamma \to \mathsf{F}\ \gamma)\ \alpha) \\
\equiv_\beta \ & \lambda\mathsf{F}{:}*{\to}*.\ \forall\alpha{:}*.\ \mathsf{F}\ (\mathsf{F}\ \alpha \to \mathsf{F}\ \alpha) \\
= \ & \overline{\forall\alpha{:}*.\ \alpha \to \alpha}
\end{aligned}$$

Properties. We now discuss some properties of our type representation that are important for representing terms. First, we can pre-represent legal types of any kind and in any environment. Since a representation abstracts over the designated type variable F in a pre-representation , the representation of a κ-type is a type of kind $(* \to *) \to \kappa$. In particular, base types (i.e. types of kind $*$) are represented by a type of kind $(* \to *) \to *$. This kind will be important for representing terms, so in Figure 2.3 we define $\mathsf{U} = (* \to *) \to *$.

Theorem 2.5.1. *If $\Gamma \vdash \tau : \kappa$, then $\Gamma \vdash \overline{\tau} : (* \to *) \to \kappa$.*

Equivalence preservation relies on the following substitution theorem, which will also be important for our representation of terms.

Theorem 2.5.2. *For any types τ and σ, and any type variable α, we have $[\![\tau]\!][\alpha := [\![\sigma]\!]] = [\![\tau[\alpha := \sigma]]\!]$.*

We now formally state the equivalence preservation property of type pre-representation and representation.

Theorem 2.5.3. *$\tau \equiv \sigma$ if and only if $\overline{\tau} \equiv \overline{\sigma}$.*

2.6 Representing Terms

In this section we describe our representation of F_ω terms. Our representations are typed to ensure that only well-typed terms can be represented. We typecheck representations of terms using type representations. In particular, a term of type τ is represented by a term of type `Exp` $\overline{\tau}$.

We use a typed Higher-Order Abstract Syntax (HOAS) representation based on Church encodings, similar to those used in previous work [69, 71, 19]. As usual in Higher-Order Abstract Syntax (HOAS), we represent variables and abstractions meta-circularly, that is, using variables and abstractions. This avoids the need to implement capture-avoiding substitution on our operations – we inherit it from the host language implementation. As in our previous work [19], our representation is also parametric (PHOAS) [92, 29]. In PHOAS representations, the types of variables are parametric. In our case, they are parametric in the type function `F` that defines an interpretation of types.

Our representation of F_ω terms is shown in Figure 2.3. We define our representation in two steps, as we did for types. The pre-representation of a term is defined using the designated variables `F`, `abs`, `app`, `tabs`, and `tapp`. The representation abstracts over these variables in the pre-representation .

While the pre-representation of types can be defined by the type alone, the pre-representation of a term depends on its typing judgment. We call the function that maps typing judgments to pre-representations the *pre-quoter*. We write $\Gamma \vdash$ `e :` $\tau \rhd$ `q` to denote "given an input judgment $\Gamma \vdash$ `e :` τ the pre-quoter outputs a pre-representation `q`". The pre-representation of a term is defined by a type function `F` that defines pre-representations of types, and by four *case functions* that together define a fold over the structure of a term. The types of each case function depends on the type function `F`. The case functions are named `abs`, `app`, `tabs`, and `tapp`, and respectively represent λ-abstraction, function application, type-abstraction, and type application.

The representation $\overline{\texttt{e}}$ of a closed term `e` is obtained by abstracting over the variables `F`, `abs`, `app`, `tabs`, and `tapp` in the pre-representation of `e`. If `e` has type τ, its pre-representation has type $\texttt{F}\ [\![\tau]\!]$, and its representation has type $\texttt{Exp}\ \overline{\tau}$. The choice of τ can be arbitrary because typings are unique up to β-equivalence and type representation preserves β-equivalence.

Stripping redundant quantifiers. In addition to the `inst` functions discussed in Section 2.4, our quoter embeds a specialized variant of instantiation functions into representations. These functions can strip redundant quantifiers, which would otherwise limit the expressiveness of our HOAS representation. For example, our `size` operation will use them to remove the redundant quantifier from intermediate values with types of the form $(\forall\alpha{:}\kappa.\texttt{Nat})$. The type `Nat` is closed, so α does not occur free in `Nat`. This is why the quantifier is said to be redundant. This problem of redundant quantifiers is well known, and applies to other HOAS representations as well [71].

We can strip a redundant quantifier with a type application: if `e` has type $(\forall\alpha{:}\kappa.\texttt{Nat})$ and σ is a type of kind κ, then $\texttt{e}\ \sigma$ has the type `Nat`. We can also use the instantiation function $\texttt{inst}_{(\forall\alpha:\kappa.\texttt{Nat}),\sigma}$, which has type $(\forall\alpha{:}\kappa.\texttt{Nat}) \to \texttt{Nat}$. The choice of σ is arbitrary – it can be any type of kind κ. It happens that in F_ω all kinds are inhabited, so we can always find an appropriate σ to strip a redundant quantifier.

Our quoter generates a single strip function for each type abstraction in a term and embeds it into the representation. At the time of quotation most quantifiers are not redundant – redundant quantifiers are introduced by certain operations like `size`. Whether a quantifier will become redundant depends on the result type function `F` for an operation. In our operations, redundant quantifiers are introduced when `F` is a constant function. The operation `size` has results typed using the constant `Nat` function $\texttt{KNat} = (\lambda\alpha{:}*.\texttt{Nat})$. Each strip function is general enough to work for multiple operations that introduce redundant quantifiers, and to still allow operations like `unquote` that need the quantifier.

To provide this generality, the strip functions take some additional inputs that help establish that a quantifier is redundant before stripping it. Each strip function will have a type of the form $\texttt{Strip F}\ [\![\forall\alpha{:}\kappa.\tau]\!] \equiv (\forall\beta{:}*.\ (\forall\gamma{:}*.\ \texttt{F}\ \gamma \to \beta) \to [\![\forall\alpha{:}\kappa.\tau]\!] \to \beta)$. The type `F` is the type function defines an interpretation of types. The type $[\![\forall\alpha{:}\kappa.\tau]\!]$ is the quantified type with the redundant quantifier to be stripped. Recall that $[\![\forall\alpha{:}\kappa.\tau]\!]$ = $(\forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!])$. The type term of type $(\forall\gamma{:}*.\ \mathtt{F}\ \gamma \to \beta)$ shows that `F` is a constant function that always returns β. The strip function uses it to turn the type $(\forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!])$ into the type $(\forall\alpha{:}\kappa.\beta)$ where α has become redundant. For `size`, we will have `F = KNat = (`$\lambda\alpha{:}*.$`Nat)`. We show that `KNat` is the constant `Nat` function with an identity function $(\Lambda\gamma{:}*.\lambda\mathtt{x}{:}\mathtt{KNat}\ \gamma.\ \mathtt{x})$. The type of this function is $(\forall\gamma{:}*.\mathtt{KNat}\ \gamma \to \mathtt{KNat}\ \gamma)$, which is equivalent to $(\forall\gamma{:}*.\mathtt{KNat}\ \gamma \to \mathtt{Nat})$.

Types of case functions. The types of the four case functions `abs`, `app`, `tabs`, and `tapp` that define an interpretation, respectively `Abs`, `App`, `TAbs`, and `TApp`, are shown in Figure 2.3. The types of each function rely on invariants about pre-representations of types. For example, the type `App F` uses the fact that the pre-representation of an arrow type $[\![\tau_1 \to \tau_2]\!]$ is equal to $\mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$. In other words, `App F` abstracts over the types $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ that can change, and makes explicit the structure $\mathtt{F}\ \alpha \to \mathtt{F}\ \beta$ that is invariant. These types allow the implementation of each case function to use this structure – it is part of the "interface" of representations, and plays an important role in the implementation of each operation.

Building representations. The first rule of pre-representation handles variables. As in our type representation, variables are represented meta-circularly, that is, by other variables. We will re-use the variable name, but change its type: a variable of type τ is represented by a variable of type $\mathtt{F}\ [\![\tau]\!]$. This type is the same as the type of a pre-representation . In other words, variables in a pre-representation range over pre-representations .

The second rule of pre-representation handles λ-abstractions. We recursively pre-quote the body, in which a variable `x` can occur free. Since variables are represented meta-circularly, `x` can occur free in the pre-representation `q` of the body. Therefore, we bind `x` in the pre-representation . This is standard for Higher-Order Abstract Syntax representations. Again, we change of the type of `x` from τ_1 to $\mathtt{F}\ [\![\tau_1]\!]$. It may be helpful to think of `q` as the "open pre-representation of `e`", in the sense that `x` can occur free, and to think of $(\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q})$ as the "closed pre-representation of `e`". The open pre-representation of `e` has type $\mathtt{F}\ [\![\tau_2]\!]$ in an environment that assigns `x` the type $\mathtt{F}\ [\![\tau_1]\!]$. The closed pre-representation of `e` has type `F`

$[\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$. The pre-representation of $(\lambda\mathtt{x}{:}\tau_1.\ \mathtt{e})$ is built by applying the case function `abs` to the types $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ and the closed pre-representation of `e`.

The third rule of pre-representation handles applications. We build the pre-representation of an application $\mathtt{e}_1\ \mathtt{e}_2$ by applying the case function `app` to the types $[\![\tau_2]\!]$ and $[\![\tau]\!]$ and the pre-representations of $\mathtt{e}_1$ and $\mathtt{e}_2$.

The fourth rule of pre-representation handles type abstractions. As for λ-abstractions, we call `q` the open pre-representation of `e`, and abstract over α to get the closed pre-representation of `e`. Unlike for λ-abstractions, we do not pass the domain and codomain of the type to the case function `tabs`, since that would require kind-polymorphism as discussed in Section 2.4. Instead, we pass to `tabs` the pre-representation of the quantified type directly. We also pass to `tabs` a quantifier stripping function that enables `tabs` to remove the quantifier from $[\![\forall\alpha{:}\kappa.\ \mathtt{F}\ \tau]\!]$ in case `F` is a constant function. Note that the strip function is always defined, since $[\![\forall\alpha{:}\kappa.\ \mathtt{F}\ \tau]\!] = \forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!]$.

The fifth rule of pre-quotation handles type applications. We build the pre-representation of a type application $\mathtt{e}\ \sigma$ by applying the case function `tapp` to the pre-representation of the quantified type $[\![\forall\alpha{:}\kappa.\tau]\!]$, the pre-representation of the term `e`, the pre-representation of the instantiation type $[\![\tau[\alpha{:=}\sigma]]\!]$, and the instantiation function $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau]\!],[\![\sigma]\!])}$, which can apply any term of type $[\![\forall\alpha{:}\kappa.\tau]\!]$ to the type $[\![\sigma]\!]$. Since $[\![\forall\alpha{:}\kappa.\tau]\!] = (\forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!])$, the instantiation function has type $[\![\forall\alpha{:}\kappa.\tau]\!] \to \mathtt{F}\ [\![\tau[\alpha{:=}\sigma]]\!]$.

The last rule of pre-quotation handles the type-conversion rule. Unsurprisingly, the pre-representation of `e` is the same when `e` has type σ as when it has type τ. When `e` has type τ, its pre-representation will have type $\mathtt{F}\ [\![\tau]\!]$. When `e` has type σ, its pre-representation will have type $\mathtt{F}\ [\![\sigma]\!]$. By Theorem 2.5.3, these two types are equivalent, so `q` can be given either type.

Examples. We now give two example representations. Our first example is the representation of the polymorphic identity function $\Lambda\alpha{:}*.\lambda\mathtt{x}{:}\alpha.\mathtt{x}$:

```
ΛF:∗ → ∗.
λabs:Abs F. λapp:App F.
λtabs:TAbs F. λtapp:TApp F.
tabs ⟦∀α:∗. α → α⟧ strip_(F,∗,⟦λα:∗.α→α⟧)
  (Λα:∗. abs α α (λx:F α. x))
```

We begin by abstracting over the type function F that defines an interpretation of types, and the four case functions that define an interpretation of terms. Then we build the pre-representation of $\Lambda\alpha{:}{*}.\lambda x{:}\alpha.x$. We represent the type abstraction using `tabs`, the term abstraction using `abs`, and the variable `x` as another variable also named `x`.

Our second example is representation of $(\lambda x{:}(\forall\alpha{:}{*}.\ \alpha \to \alpha).\ x\ (\forall\alpha{:}{*}.\ \alpha \to \alpha)\ x)$, which applies an input term to itself.

```
ΛF:∗ → ∗.
λabs:Abs F. λapp:App F.
λtabs:TAbs F. λtapp:TApp F.
abs ⟦∀α:∗. α → α⟧ ⟦∀α:∗. α → α⟧
 (λx: F ⟦∀α:∗. α → α⟧.
  app ⟦∀α:∗. α → α⟧ ⟦∀α:∗. α → α⟧
    (tapp ⟦∀α:∗. α → α⟧ x
          ⟦(∀α:∗. α → α) → (∀α:∗. α → α)⟧
          inst_(⟦∀α:∗.α→α⟧,⟦∀α:∗.α→α⟧))
    x)
```

The overall structure is similar to above: we begin with the five abstractions that define interpretations of types and terms. We then use the case functions to build the pre-representation of the term. The instantiation function $\mathtt{inst}_{(\llbracket\forall\alpha:*.\alpha\to\alpha\rrbracket,\llbracket\forall\alpha:*.\alpha\to\alpha\rrbracket)}$ has the type $\llbracket\forall\alpha{:}{*}.\ \alpha \to \alpha\rrbracket \to \mathtt{F}\ \llbracket(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)\rrbracket$. Here, the quantified type being instantiated is $\llbracket\forall\alpha{:}{*}.\ \alpha \to \alpha\rrbracket = \forall\alpha{:}{*}.\ \mathtt{F}\ \llbracket\alpha \to \alpha\rrbracket$, the instantiation parameter is also $\llbracket\forall\alpha{:}{*}.\ \alpha \to \alpha\rrbracket$, and the instantiation type is $\mathtt{F}\ \llbracket(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)\rrbracket$. By lemma 2.5.2, we have:

$$
\begin{aligned}
&(\mathtt{F}\ \llbracket\alpha \to \alpha\rrbracket)[\alpha := \llbracket\forall\alpha{:}{*}.\ \alpha \to \alpha\rrbracket]\\
=&\mathtt{F}\ (\llbracket\alpha \to \alpha\rrbracket[\alpha := \llbracket\forall\alpha{:}{*}.\ \alpha \to \alpha\rrbracket])\\
=&\mathtt{F}\ \llbracket(\alpha \to \alpha)[\alpha := \forall\alpha{:}{*}.\ \alpha \to \alpha]\rrbracket\\
=&\mathtt{F}\ \llbracket(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)\rrbracket
\end{aligned}
$$

Properties. We typecheck pre-quotations under a modified environment that changes the types of term variables and binds the variables `F`, `abs`, `app`, `tabs`, and `tapp`. The bindings of type variables are unchanged.

The environment for pre-quotations of closed terms only contains bindings for `F`, `abs`, `app`, `tabs`, and `tapp`. The representation of a closed term abstracts over these variables, and so can be typed under an empty environment.

Theorem 2.6.1. *If* $\langle\rangle \vdash \mathtt{e} : \tau$*, then* $\langle\rangle \vdash \overline{\mathtt{e}} : \mathtt{Exp}\ \overline{\tau}$*.*

Our representations are *data*, which for F_ω means a β-normal form.

Theorem 2.6.2. *If* $\langle\rangle \vdash \mathtt{e} : \tau$*, then* $\overline{\mathtt{e}}$ *is* β*-normal.*

Our quoter preserves equality of terms up to equivalence of types. That is, if two terms are equal up to equivalence of types, then their representations are equal up to equivalence of types as well. Our quoter is also injective up to equivalence of types, so the converse is also true: if the representations of two terms are equal up to equivalence of types, then the terms are themselves equal up to equivalence of types.

Definition 2.6.1 (Equality up to equivalence of types)**.** *We write* $\mathtt{e}_1 \sim \mathtt{e}_2$ *to denote that terms* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are equal up to equivalence of types.*

$$\mathtt{x} \sim \mathtt{x}$$

$$\frac{\tau \equiv_\beta \tau' \qquad \mathtt{e} \sim \mathtt{e}'}{(\lambda\mathtt{x}{:}\tau.\mathtt{e}) \sim (\lambda\mathtt{x}{:}\tau'.\mathtt{e}')} \qquad\qquad \frac{\mathtt{e}_1 \sim \mathtt{e}_1' \qquad \mathtt{e}_2 \sim \mathtt{e}_2'}{(\mathtt{e}_1\ \mathtt{e}_2) \sim (\mathtt{e}_2'\ \mathtt{e}_2')}$$

$$\frac{\mathtt{e} \sim \mathtt{e}'}{(\Lambda\alpha{:}\kappa.\mathtt{e}) \sim (\Lambda\alpha{:}\kappa.\mathtt{e}')} \qquad\qquad \frac{\mathtt{e} \sim \mathtt{e}' \qquad \tau \equiv_\beta \tau'}{(\mathtt{e}\ \tau) \sim (\mathtt{e}'\ \tau')}$$

Now we can formally state that our quoter is injective up to equivalence of types.

Theorem 2.6.3. *If* $\langle\rangle \vdash \mathtt{e}_1 : \tau$*, and* $\langle\rangle \vdash \mathtt{e}_2 : \tau$*, then*
$\mathtt{e}_1 \sim \mathtt{e}_2$ *if and only if* $\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}$*.*

```
Bool  : *    = ∀α:*. α → α → α
true  : Bool = Λα:*. λt:α. λf:α. t
false : Bool = Λα:*. λt:α. λf:α. f
and   : Bool → Bool → Bool =
  λb1:Bool. λb2:Bool. Λα:*. λt:α. λf:α.
  b1 α (b2 α t f) f

Bools : * = ∀α:*. (Bool → Bool → α) → α
bools : Bool → Bool → Bools =
  λb1:Bool. λb2:Bool.
  Λα:*. λf:Bool → Bool → α. f b1 b2
fst : Bools → Bool =
  λbs:Bools. bs Bool (λb1:Bool. λb2:Bool. b1)
snd : Bools → Bool =
  λbs:Bools. bs Bool (λb1:Bool. λb2:Bool. b2)

Nat  : *   = ∀α:*. α → (α → α) → α
zero : Nat = Λα:*. λz:α. λs:α → α. z
succ : Nat → Nat =
  λn:Nat. Λα:*. λz:α. λs:α → α. s (n α z s)
plus : Nat → Nat → Nat =
  λm:Nat. λn:Nat. m Nat n succ
```

Definitions and operations of `Bool`, `Bools`, and `Nat`.

```
foldExp : (∀F:∗ → ∗.
           Abs F → App F → TAbs F → TApp F →
           ∀α:U. Exp α → Op F α) =
  ΛF:∗ → ∗.
  λabs  : Abs F.  λapp  : App F.
  λtabs : TAbs F. λtapp : TApp F.
  Λα:U. λe:Exp α. e F abs app tabs tapp
```

Implementation of `foldExp`

```
Id  : ∗ → ∗ = λα:∗.α

unAbs : Abs Id = Λα:∗.Λβ:∗.λf:α→β.f
unApp : App Id = Λα:∗.Λβ:∗.λf:α→β.λx:α.f x
unTAbs : TAbs Id = Λα:∗.λs:Strip Id α.λf:α.f
unTApp : TApp Id = Λα:∗.λf:α.Λβ:∗.λg:α→β.g f

unquote : (∀α:U. Exp α → Op Id α) =
  foldExp Id unAbs unApp unTAbs unTApp
```

Implementation of `unquote`

```
KBool : ∗ → ∗ = λα:∗. Bool

isAbsAbs : Abs KBool =
  Λα:∗. Λβ:∗. λf:Bool → Bool. true
isAbsApp : App KBool =
  Λα:∗. Λβ:∗. λf:Bool. λx:Bool. false
isAbsTAbs : TAbs KBool =
  Λα:∗. λstrip:Strip KBool α. λf:α. true
isAbsTApp : TApp KBool =
  Λα:∗. λf:Bool. Λβ:∗. λinst:α → Bool. false

isAbs : (∀α:U. Exp α → Bool) =
  foldExp KBool isAbsAbs isAbsApp
                isAbsTAbs isAbsTApp
```

Implementation of `isAbs`.

```
KNat : ∗ → ∗ = λα:∗. Nat

sizeAbs : Abs KNat =
  Λα:∗. Λβ:∗. λf:Nat → Nat. succ (f (succ zero))
sizeApp : App KNat =
  Λα:∗. Λβ:∗. λf:Nat. λx:Nat. succ (plus f x)
sizeTAbs : TAbs KNat =
  Λα:∗. λstrip:Strip KNat α. λf:α.
  succ (strip Nat (Λα:∗. λx:Nat. x) f)
sizeTApp : TApp KNat =
  Λα:∗. λf : Nat. Λβ:∗. λinst:α → Nat. succ f

size : (∀α:U. Exp α → Nat) =
  foldExp KNat sizeAbs sizeApp sizeTAbs sizeTApp
```

Implementation of `size`.

```
KBools : ∗ → ∗ = λα:∗. Bools

nfAbs : Abs KBools =
  Λα:∗. Λβ:∗. λf:Bools → Bools.
  bools (fst (f (bools true true))) false
nfApp : App KBools =
  Λα:∗. Λβ:∗. λf:Bools. λx:Bools.
  bools (and (snd f) (fst x)) (and (snd f) (fst x))
nfTAbs : TAbs KBools =
  Λα:∗. λstrip:Strip KBools α. λf:α.
  bools (fst (strip Bools (Λα:∗.λx:Bools.x) f))
        false
nfTApp : TApp KBools =
  Λα:∗. λf:Bools. Λβ:∗. λinst:(α → Bools).
  bools (snd f) (snd f)

nf : (∀α:U. Exp α → Bool) =
  Λα:U. λe:Exp α.
  fst (foldExp KBools nfAbs nfApp nfTAbs nfTApp e)
```

Implementation of `nf`.

```
Ct  : ∗ → ∗ = λα:∗. ∀β:∗. (α → β) → β
CPS : U → ∗ = Op Ct

cpsAbs : Abs Ct =
  Λα:∗. Λβ:∗. λf:(Ct α → Ct β).
  ΛV:∗. λk : (Ct α → Ct β) → V.
  k f
cpsApp : App Ct =
  Λα:∗. Λβ:∗. λf:Ct (Ct α → Ct β). λx:Ct α.
  ΛV:∗. λk:β → V.
  f V (λg:Ct α → Ct β. g x V k)
cpsTAbs : TAbs Ct =
  Λα:∗. λstrip:Strip Ct α. λf: α.
  ΛV:∗. λk:α → V.
  k f
cpsTApp : TApp Ct =
  Λα:∗. λf: Ct α.
  Λβ:∗. λinst:α → Ct β.
  ΛV:∗. λk:β → V.
  f V (λe:α. inst e V k)

cps : (∀α:U. Exp α → CPS α) =
  foldExp Ct cpsAbs cpsApp cpsTAbs cpsTApp
```

Implementation of `cps`.

Figure 2.4: Five operations on representations of F_ω terms.

2.7 Operations

Our suite of operations is given in Figure 2.4. It consists of a self-interpreter `unquote`, a simple intensional predicate `isAbs`, a size measure `size`, a normal-form checker `nf`, and a continuation-passing-style transformation `cps`. Our suite extends those of each previous work on typed self-representation[71, 19]. Rendel et al. define a self-interpreter and a size measure, while in previous work we defined a self-interpreter, the intensional predicate `isAbs`, and a CPS transformation. Our normal-form checker is the first for a typed self-representation.

Each operation is defined using a function `foldExp` for programming folds. We also define encodings of booleans, pairs of booleans, and natural numbers that we use in our operations. We use a declaration syntax for types and terms. For example, the term declaration $\mathtt{x} : \tau = \mathtt{e}$ asserts that `e` has the type τ (i.e. $\langle\rangle \vdash \mathtt{e} : \tau$ is derivable), and substitutes `e` for `x` (essentially inlining `x`) in the subsequent declarations. We have machine checked the type of each declaration.

We give formal semantic correctness proofs for four of our operations: `unquote`, `isAbs`, `size`, and `nf`. The proofs demonstrate qualitatively that our representation is not only expressive but also easy to reason with. In the remainder of this section we briefly discuss the correctness theorems.

Each operation has a type of the form $\forall\alpha$`:U. Exp` $\alpha \rightarrow$ `Op R` α for some type function `R`. When α is instantiated with a type representation $\overline{\tau}$, the result type `Op R` $\overline{\tau}$ is an interpretation under `R`:

Theorem 2.7.1. `Op R` $\overline{\tau} \equiv$ `R (`$[\![\tau]\!]$`[F := R])`.

Each operation is defined using the function `foldExp` that constructs a fold over term representations. An interpretation of a term is obtained by substituting the designated variables `F`, `abs`, `app`, `tabs`, and `tapp` with the case functions that define an operation. The following theorem states that a fold constructed by `foldExp` maps representations to interpretations:

Theorem 2.7.2. *If* `f = foldExp R abs′ app′ tabs′ tapp′`*, and* $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$*, then* `f` $\overline{\tau}$ $\overline{\mathtt{e}}$.

$\longrightarrow^*$ `(q[F:=R, abs:=abs′,`
`app:=app′, tabs:=tabs′, tapp:=tapp′])`.

unquote. Our first operation on term representations is our self-interpreter `unquote`, which recovers a term from its representation. Its results have types of the form `Op Id` $\overline{\tau}$. The type function `Id` is the identity function, and the operation `Op Id` recovers a type from its representation.

Theorem 2.7.3. *If* $\Gamma \vdash \tau : *$*, then* `Op Id` $\overline{\tau} \equiv \tau$.

Notice that `unquote` has the polymorphic type `(`$\forall\alpha$`:U. Exp` $\alpha \rightarrow$ `Op Id` α`)`. The type variable α ranges over representations of types, and the result type `Op Id` α recovers the type α represents. Thus, when α is instantiated with a concrete type representation $\overline{\tau}$, we get the type `Exp` $\overline{\tau} \rightarrow \tau$.

Theorem 2.7.4. *If* $\langle\rangle \vdash$ `e` $: \tau$*, then* `unquote` $\overline{\tau}$ $\overline{\texttt{e}} \longrightarrow^*$ `e`.

isAbs. Our second operation `isAbs` is a simple intensional predicate that checks whether its input represents an abstraction or an application. It returns a boolean on all inputs. Its result types are interpretations under `KBool`, the constant `Bool` function. The interpretation of any type under `KBool` is equivalent to `Bool`:

Theorem 2.7.5. *If* $\Gamma \vdash \tau : *$*, then* `Op KBool` $\overline{\tau} \equiv$ `Bool`.

Theorem 2.7.6. *Suppose* $\langle\rangle \vdash$ `e` $: \tau$*. If* `e` *is an abstraction then* `isAbs` $\overline{\tau}$ $\overline{\texttt{e}} \longrightarrow^*$`true`*. Otherwise* `e` *is an application and* `isAbs` $\overline{\tau}$ $\overline{\texttt{e}} \longrightarrow^*$`false`.

size. Our third operation `size` measures the size of its input representation. Its result types are interpretations under `KNat`, the constant `Nat` function. The interpretation of any type under `KNat` is equivalent to `Nat`:

Theorem 2.7.7. *If* $\Gamma \vdash \tau : *$*, then* `Op KNat` $\overline{\tau} \equiv$ `Nat`.

The size of a term excludes the types. We formally define the size of a term in order to state the correctness of `size`.

Definition 2.7.1. *The size of a term* `e`*, denoted* `|e|`*, is defined as:*

$$\begin{array}{ll}
|\mathtt{x}| & = 1 \\
|\lambda \mathtt{x}{:}\tau.\mathtt{e}| & = 1 + |\mathtt{e}| \\
|\mathtt{e}_1\ \mathtt{e}_2| & = 1 + |\mathtt{e}_1| + |\mathtt{e}_2| \\
|\Lambda\alpha{:}\kappa.\mathtt{e}| & = 1 + |\mathtt{e}| \\
|\mathtt{e}\ \tau| & = 1 + |\mathtt{e}|
\end{array}$$

The results of `size` are Church encodings of natural numbers. We define a type `Nat` and a `zero` element and a successor function `succ`. We use the notation $\mathtt{church}_n$ to denote the Church-encoding of the natural number n. For example, $\mathtt{church}_0$ `= zero`, $\mathtt{church}_1$ `= succ zero`, $\mathtt{church}_2$ `= succ (succ zero)`, and so on.

Theorem 2.7.8.
If $\langle\rangle \vdash \mathtt{e} : \tau$ *and* $|\mathtt{e}|$=n*, then* $\mathtt{size}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^{*} \mathtt{church}_n$

nf. Our fourth operation `nf` checks whether its input term is in β-normal form. Its results have types that are interpretations under `KBools`, the constant `Bools` function, where `Bools` is the type of pairs of boolean values.

Theorem 2.7.9. *If* $\Gamma \vdash \tau : *$*, then* $\mathtt{Op\ KBools}\ \overline{\tau} \equiv \mathtt{Bools}$.

We program `nf` in two steps: first, we compute a pair of booleans by folding over the input term. Then we return the first component of the pair. The first boolean encodes whether a term is β-normal. The second encodes whether a term is normal and *neutral*. Intuitively, a neutral term is one that can be used in function position of an application without introducing a redex. We provide a formal definition of normal and neutral in the Appendix.

Theorem 2.7.10. *Suppose* $\langle\rangle \vdash \mathtt{e} : \tau$.

1. *If* `e` *is β-normal, then* $\mathtt{nf}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^{*} \mathtt{true}$.

2. *If* `e` *is not β-normal, then* $\mathtt{nf}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^{*} \mathtt{false}$.

cps. Our fifth operation `cps` is a call-by-name continuation-passing-style transformation. Its result types are interpretations under `Ct`. We have also implemented a call-by-value CPS transformation, though we omit the details because it is rather similar to our call-by-name CPS. We do not formally prove the correctness of our CPS transformation. However, being defined in F_ω it is guaranteed to terminate for all inputs, and the types of the case functions provide some confidence in its correctness. Below, we show the result of applying `cps` to each of the example representations from Section 2.6. To aid readability, we use $[\![\tau]\!]_{\texttt{Ct}}$ to denote $[\![\tau]\!]$`[F := Ct]`.

The first example is the polymorphic identity function $\Lambda\alpha{:}*.\ \lambda\texttt{x}{:}\alpha.\ \texttt{x}$:

$$
\begin{array}{l}
\quad\texttt{cps}\ \overline{\forall\alpha{:}*.\ \alpha\to\alpha}\ \overline{\Lambda\alpha{:}*.\lambda\texttt{x}{:}\alpha.\texttt{x}} \\[1em]
\equiv_\beta \texttt{cpsTAbs}\ [\![\forall\alpha{:}*.\ \alpha\to\alpha]\!]_{\texttt{Ct}}\ \texttt{strip}_{\texttt{Ct},[\![\forall\alpha:*.\alpha\to\alpha]\!]} \\
\quad\quad (\Lambda\alpha{:}*.\ \texttt{cpsAbs}\ \alpha\ \alpha\ (\lambda\texttt{x}{:}\texttt{Ct}\ \alpha.\ \texttt{x})) \\[1em]
\equiv_\beta \Lambda\beta_1 : *. \\
\quad \lambda\texttt{k}_1 : \texttt{Ct}\ (\forall\alpha{:}*.\ \texttt{Ct}\ (\texttt{Ct}\ \alpha\to\texttt{Ct}\ \alpha))\to\beta_1. \\
\quad \texttt{k}_1\ (\Lambda\alpha{:}*.\ \Lambda\beta_2{:}*. \\
\quad\quad\quad \lambda\texttt{k}_2 : (\texttt{Ct}\ \alpha\to\texttt{Ct}\ \alpha)\to\beta_2. \\
\quad\quad\quad \texttt{k}_2\ (\lambda\texttt{x} : \texttt{Ct}\ \alpha.\ \texttt{x}))
\end{array}
$$

The second example applies a variable of type $\forall\alpha{:}*.\alpha\to\alpha$ to itself:

$$\begin{array}{l}
\quad\mathtt{cps}\ \overline{(\forall\alpha{:}*.\ \alpha \to \alpha) \to (\forall\alpha{:}*.\ \alpha \to \alpha)} \\
\qquad\overline{\lambda\mathtt{x}{:}(\forall\alpha{:}*.\ \alpha \to \alpha).\mathtt{x}\ (\forall\alpha{:}*.\ \alpha \to \alpha)\ \mathtt{x}} \\
\equiv_\beta \mathtt{cpsAbs}\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]_{\mathtt{Ct}} \\
\qquad(\lambda\mathtt{x}{:}\ \mathtt{Ct}\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}. \\
\qquad\ \ \mathtt{cpsApp}\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]_{\mathtt{Ct}} \\
\qquad\qquad(\mathtt{cpsTApp}\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ \mathtt{x} \\
\qquad\qquad\qquad\quad[\![(\forall\alpha{:}*.\ \alpha \to \alpha) \to (\forall\alpha{:}*.\ \alpha \to \alpha)]\!]_{\mathtt{Ct}} \\
\qquad\qquad\qquad\quad\mathtt{inst}_{[\![\forall\alpha:*.\alpha\to\alpha]\!],[\![\forall\alpha:*.\alpha\to\alpha]\!]}) \\
\qquad\qquad\mathtt{x})
\end{array}$$

$$\begin{array}{l}
\equiv_\beta \Lambda\beta_1\ \mathtt{:}\ *. \\
\quad\lambda\mathtt{k}_1\ \mathtt{:}\ (\mathtt{Ct}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a})) \to \\
\qquad\qquad\mathtt{Ct}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a}))) \to \beta_1. \\
\quad\mathtt{k}_1\ (\\
\quad\ \lambda\mathtt{x}\ \mathtt{:}\ \mathtt{Ct}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a})). \\
\quad\ \Lambda\beta_2\ \mathtt{:}\ *. \\
\quad\ \lambda\mathtt{k}_2\ \mathtt{:}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a})) \to \beta_2. \\
\quad\ (\mathtt{x}\ \beta_2\ (\lambda\mathtt{e}\ \mathtt{:}\ \forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a}). \\
\qquad\qquad\mathtt{e}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a}))\ \beta_2 \\
\qquad\qquad\quad(\lambda\mathtt{g}\ \mathtt{:}\ \mathtt{Ct}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a})) \to \\
\qquad\qquad\qquad\qquad\mathtt{Ct}\ (\forall\mathtt{a}{:}*.\ \mathtt{Ct}\ (\mathtt{Ct\ a} \to \mathtt{Ct\ a})). \\
\qquad\qquad\qquad\mathtt{g\ x}\ \beta_2\ \mathtt{k}_2))))
\end{array}$$

2.8 Experiments

We have validated our techniques using an implementation of F_ω in Haskell, consisting of a parser, type checker, evaluator, β-equivalence checker, and our quoter. Each operation has been programmed, type checked, and tested. We have also confirmed that the representation of each operation type checks with the expected type.

Each of our operations are self-applicable, meaning it can be applied to a representation of itself. We have checked that the self-application of each operation type checks with the expected type. Further, we have checked that the self-application of `unquote` is β-equivalent to itself:

$$\begin{array}{l}
\quad\mathtt{unquote}\ \overline{(\forall\alpha{:}\mathtt{U}.\ \mathtt{Exp}\ \alpha \to \mathtt{Op\ Id}\ \alpha)}\ \overline{\mathtt{unquote}} \\
\equiv_\beta \mathtt{unquote}
\end{array}$$

2.9 Discussion

Terminology. The question of whether any language (strongly normalizing or not) supports self-interpretation depends fundamentally on how one defines "representation function" and "interpreter". There are two commonly used definitions of "interpreter". The first is a function that maps the representation of a term to the term's value (i.e. the left-inverse of a particular representation function), like `eval` in JavaScript and Lisp [59, 4, 10, 69, 13, 63, 22, 71, 30]. For clarity, we will sometimes refer to this as an unquoter, since the representation function is often called a quoter. Our self-interpreter in particular is an unquoter. The other possible definition is a function that maps a representation of a term to the *representation* of its value. In other words, it implements evaluation on representations. For this reason, we call this kind of interpreter an evaluator. Evaluators are sometimes simply called interpreters [72, 66] but other names have been used to differentiate them from unquoters: Mogensen calls them reducers [63], Berarducci and Böhm call them reductors [13], and Jay and Palsberg call them enactors [52].

Some qualitative distinctions between interpreters can also be made, which might also affect the possibility of self-interpretation. A notable example is whether an interpreter is meta-circular. Unfortunately, the term "meta-circular" also has multiple meanings. Reynolds defines a meta-circular interpreter to be one which "defines each feature of the defined language by using the corresponding feature of the defining language" [72]. Abelson and Sussman state "an evaluator that is written in the same language that it evaluates is said to be metacircular" [4]. Here "evaluator" is used to mean an unquoter. Reynolds' definition allows for meta-circular interpreters that are not self-interpreters, and self-interpreters that are not meta-circular. According to Abelson and Sussman, all self-interpreters are meta-circular and vice versa. Our self-interpreter is meta-circular according to both definitions.

Many different representation schemes have been used to define interpreters. Terms can be represented as numbers, S-expressions, Church-encodings, or some user-defined data type. With respect to the treatment of variables, there is first-order, higher-order, and parametric higher-order abstract syntax. For representations of statically typed languages, use of typed representation ensures that only well-typed terms can be represented. Typed representation uses a family of types for representations, indexed by the type of the represented term, while untyped representation uses a single type for all representations. We use a typed parametric higher-order abstract syntax representation based on Church encoding. As far as we know, all unquoters defined in statically typed meta-languages are based on typed representation. Indeed, typed representation seems to be required to define an unquoter in a statically typed meta-language.

There are some properties common to all of these representation schemes: the representation function must be total (all legal terms can be represented) and injective (two terms are identical if and only if their representations are), and must produce data (e.g. normal forms). These are the requirements we have used in this chapter. It is possible to strengthen the definition further. For example, we might want to require that a representation be deep. In such a case, only our deep representation would qualify as a representation.

Deep and Shallow Representation. We use the terms deep and shallow to differentiate representations supporting multiple operations (interpretations) from those supporting only one. This is analogous to deep versus shallow *embeddings* [43], but we emphasize a key difference between representation and embedding: a representation is required to be a normal form. This is generally not required of embeddings; indeed, shallow embeddings typically do not produce normal forms. A shallow embedding translates a term in one language into its interpretation defined in another language. In summary, a shallow representation *supports* one interpretation, while a shallow embedding *is* one interpretation.

Every language trivially supports shallow self-embedding, but the same is not true for shallow self-representation. For example, a shallow self-embedding for the simply-typed lambda calculus can simply map each term to itself. This is not a self-representation because the result may not be a normal form. The shallow self-representation in Section 3.3 relies on polymorphism, so it would not would work for simply typed lambda calculus.

Limitations. There are some limits to the operations we can define on our representation. For example, we cannot define our representation functions (Figure 3) in F_ω itself. This is necessarily so because the representation functions are intensional and F_ω is extensional. Stump [81] showed that it is possible to define a self-representation function within a λ-calculus extended with some intensional operations. There is a trade-off between extensional and intensional calculi: intensionality can support more expressive meta-programming, but extensionality is important for semantic properties like parametricity.

Another limitation is that representations need to be statically type checked, which limits dynamic generation of representations. For example, it is unlikely we could implement a ”dynamic type checker” that maps an untyped representation to a typed representation. Something similar may be possible using stages interleaved with type checking, where an untyped representation in one stage becomes a typed representation in the next. Kiselyov calls this “Metatypechecking” [56].

Universe Hierarchies. Our techniques can be used for self-representation of other strongly-normalizing calculi more powerful than F_ω. For example, we conjecture that using kind-instantiation functions could enable a deep self-representation of F^+_ω. We would only need to use the extensional approach for representing kind polymorphism, since the kind polymorphism of F^+_ω would enable the intensional approach to representing type polymorphism. More generally, our techniques could be used to represent a language with a hierarchy of n universes of types, with lowest universe being impredicative and the others predicative. We could use the intensional approach for the lowest $n-1$ universes, and tie the knot of self-representation by using the extensional approach for the n^{th} universe.

Type Equivalence. Our formalization of F_ω supports type conversion based on β-equivalence. In other words, two types are considered equivalent if they are beta-equivalent. This is standard – Barendregt[11] and Pierce[70] each use β-equivalence as well. Our representation would also work with a type conversion rule based on β,η-equivalence. Our key theorems 2.6.1 and 2.6.2 would be unaffected, and Theorem 2.6.3 would also hold assuming we update Definition 2.6.1 (equality of terms up to equivalence of types) accordingly.

Injective Representation Function. Intuitively, our result shows that our quoter is injective because it has a left inverse unquote. In practice, we proved injectivity in Theorem

6.3 before we went on to define unquote in Section 7. The reason is that we used injectivity to prove the correctness of unquote. We leave to future work to first define and prove correctness of unquote and then use that to prove that our quoter is injective.

2.10 Related Work

Typed Self-Interpretation. Pfenning and Lee [69] studied self-interpretation of Systems F and F_ω. They concluded that it seemed to be impossible for each language, and defined representations and unquoters of System F in F_ω and of F_ω in F_ω^+. They used the intensional approach to representing polymorphism that discussed in Section 2.4.

Rendel, et al. [71] presented the first typed self-representation and self-interpreter. Their language System F_ω^* extends F_ω with a `Type:Type` rule that unifies the levels of types and kinds. As a result, F_ω^* is not strongly-normalizing, and type checking is undecidable. Their representation also used the intensional approach to representing polymorphism. They presented two operations, an unquoter and a size measure. Their implementation of `size` relied on a special $\perp$ type to strip redundant quantifiers. The type $\perp$ inhabits every kind, but is not used to type check terms. We strip redundant quantifiers using special instantiation functions that are generated by the quoter.

Jay and Palsberg [52] presented a typed self-representation and self-interpreter for a combinator calculus, with a λ-calculus surface syntax. Their calculus had undecidable type checking and was not strongly normalizing.

In previous work [19] we presented a typed self-representation for System U, which is not strongly normalizing but does have decidable type checking. This was the first self-representation for a language with decidable type checking. The representation was similar to those of Pfenning and Lee [69] and Rendel, et al. [71] and also used the intensional approach to representing polymorphism. We presented three operations on the representation of System U terms – `unquote`, `isAbs`, and `cps`. Not all System U kinds are inhabited, so redundant quantifiers couldn't be stripped. This prevented operations like `size` or `nf`. We also represented System U types of kind $*$, but did not have a substitution theorem like

Theorem 2.5.2. As a result, the representation of a type application could have the wrong type, which we corrected using a kind of coercion. Our representation of F_ω types is designed to avoid the need for such coercions, which simplifies our representation and the proofs of our theorems.

Representation Technique. We mix standard representation techniques with a minimal amount of novelty needed to tie the knot of self-representation. At the core is a typed Higher-order Abstract Syntax (HOAS) based on Church encoding. Similar representations were used in previous work on typed representation [69, 22, 71, 19].

Our previous work [19] showed self-representation is possible using only the intensional approach to representing polymorphism requires and two impredicative universes (the types and kinds of System U). Our extensional approach presented here allows us to use only a single impredicative universe (the types of F_ω).

The shallow representation of System F also requires impredicativity to block type applications. We leave the question whether self-representation is possible without any impredicativity for future work.

Typed Meta-Programming. Typed self-interpretation is a particular instance of typed meta-programming, which involves a typed representation of one language in a possibly different language, and operations on that representation. Typed meta-programming has been studied extensively, and continues to be an active research area. Chen and Xi [24, 25] demonstrated that types can make meta-programming less error-prone.

Carette et al. [22] introduced tagless representations, which are more efficient than other techniques and use simpler types. Our representation is also tagless, though we use ordinary λ-abstractions to abstract over the case functions of an operation, while they use Haskell type classes or OCaml modules. The object languages they represented did not include polymorphism. Our extensional technique could be used to program tagless representations of polymorphic languages in Haskell or OCaml.

MetaML [84] supports *generative* typed meta-programming for multi-stage programming. It includes a built-in unquoter, while we program `unquote` as a typed F_ω term.

Trifonov et al. [86] define a language with fully reflexive intensional type analysis, which supports type-safe run-time type introspection. Instead of building representations of types, their language includes special operators to support iterating over types. They programmed generic programs like marshalling values for transmission over a network. Generic programming and meta-programming are different techniques: generic programs operate on programs or program values, and meta-programs operate on representations of programs. These differences mean that each technique is suited to some problems better than the other.

Dependently-Typed Representation. Some typed representations use dependent types to ensure that only well-typed terms can be represented. For example, Harper and Licata [49] represented simply-typed λ-calculus in LF, and Schürmann et al. [76] represented F_ω in LF. Chapman [23] presented a meta-circular representation of a dependent type theory in Agda. These representations are quite useful for *mechanized meta-theory* – machine-checked proofs of the meta-theorems for the represented language. The demands of mechanized metatheory appear to be rather different from those of self-interpretation. It is an open question whether a dependently-typed self-representation can support a self-interpreter.

Dependent Types. Dependent type theory is of particular interest among strongly-normalizing languages, as it forms the basis of proof assistants like Coq and Agda. While dependent type theory generally includes dependent sum and product types, modern variants also support inductive definitions, an infinite hierarchy of universes, and universe polymorphism. A self-representation of such a language would need to represent all of these features, each of which comes with its own set of challenges.

Altenkirch and Kaposi [7] formalize a simple dependent type theory in another type theory (Agda extended with some postulates). They focus on the key problem of defining a typed representation of dependent type theory: that the types, terms, type contexts, and type equality are all mutually-dependent. Their solution relies on Quotient-Inductive Types (QITs), a special case of Higher-Inductive Types from Homotopy Type Theory. Their work is an important step towards a self-representation of dependently type theory. To achieve full self-representation, one would need to represent QITs themselves, which the authors cite as an open challenge.

Untyped Representation. The literature contains many examples of untyped representations for typed languages, including for Coq [12] and Haskell [66]. Untyped representations generally use a single type like `Exp` to type check all representations, and permit ill-typed terms to be represented. Template Haskell [78] uses an untyped representation and supports user-defined operations on representations. Since representations are not guaranteed to be well-typed by construction, generated code needs to be type checked.

Coercions. Our instantiation functions are similar to coercions or *retyping functions*: they change the type of a term without affecting its behavior. Cretin and Rémy [35] studied erasable coercions for System F_η [61], including coercions that perform instantiations. We conjecture that our self-representation technique would work for an extension of F_ω with erasable instantiation coercions, and that these coercions could replace instantiation functions in our extensional approach to representing polymorphism. This could provide some advantages over the instantiation functions used in this chapter. In particular, a weakness of instantiation functions is that their types overlap with those of other terms. Therefore, it is possible to use something other than instantiation function (e.g. a constant function) where one is expected. As a result, we can write a closed term of type `Exp` τ (for some τ) that is not the representation of any term. The types of Cretin and Rémy's coercions do not overlap with the types of terms, so replacing instantiation functions with instantiation coercions could eliminate this problem.

2.11 Conclusion

We have solved two open problems posed by Pfenning and Lee. First, we have defined a shallow self-representation technique that supports self-interpretation for each of System F and System F_ω. Second, we have defined a deep self-representation for System F_ω that supports a variety of operations including a self-interpreter.

Our result is consistent with the classical theorem that the universal function for the total computable functions cannot be total. The reason is that the theorem assumes that terms are represented as numbers using Gödel numbering. We show that a typed representation can ensure that the diagonalization gadget central to the proof fails to type check.

Our result opens the door to self-representations and self-interpreters for other strongly normalizing languages. Our techniques create new opportunities for type-checking self-applicable meta-programs, with potential applications in typed macro systems, partial evaluators, compilers, and theorem provers.

Some open questions include:

- Is a self-evaluator possible in a strongly normalizing language?
- Is it possible to define a self-interpreter or self-evaluator using a first-order representation (for example, based on SK combinators) in a strongly normalizing language?

CHAPTER 3

Typed Self-Evaluation via Intensional Type Functions

3.1 Introduction

Many popular languages have a self-interpreter, that is, an interpreter for the language written in *itself*; examples include Haskell [66], JavaScript [39], Python [73], Ruby [95], Scheme [5], and Standard ML [74]. The use of *itself* as implementation language is cool, demonstrates expressiveness, and has key advantages. In particular, a self-interpreter enables the language designer to easily modify, extend, and grow the language [71], and do other forms of meta-programming [18].

What is the type of an interpreter that can interpret a representation of itself? The classical answer to such questions is to work with a single type for all program representations. For example, the single type could be String or it could be Syntax Tree. The single-type approach enables an interpreter to have type, say, (String → String), where the input string represents a program and where the output string represents the result. However, this approach ignores that the source program type checks, and gives no guarantee that the interpreter preserves the type of its input.

How can we do better type checking of self-interpreters? First, suppose we have a better representation scheme *quote*(·) and a type function `Exp` such that if `e : T`, then *quote*`(e) : Exp T`. This enables us to consider two polymorphic types of self-interpreters:

$$\text{(self-recognizer)} \qquad \texttt{unquote} : \forall \texttt{T}.\ \texttt{Exp T} \to \texttt{T} \qquad (1)$$

$$\text{(self-evaluator)} \qquad \texttt{eval} : \forall \texttt{T}.\ \texttt{Exp T} \to \texttt{Exp T} \qquad (2)$$

The functionality of a self-recognizer `unquote` is to recover a program from its representation, while the functionality of a self-evaluator `eval` is to evaluate the represented program and

$$\begin{tikzcd}
\overline{\mathtt{e}} \arrow[r, "\mathtt{eval}"] \arrow[d, bend left, "\mathtt{unquote}"] & \overline{\mathtt{v}} \arrow[d, bend left, "\mathtt{unquote}"] \\
\mathtt{e} \arrow[r, "\boldsymbol{evaluation}"'] \arrow[u, bend left, "\boldsymbol{quote}"] & \mathtt{v} \arrow[u, bend left, "\boldsymbol{quote}"]
\end{tikzcd}$$

Figure 3.1: Self-recognizers and self-evaluators.

produce a representation of the result. The relationship between a self-recognizer and a self-evaluator is illustrated in Figure 3.1. The meta-level function quote maps a term `e` to its representation $\overline{\mathtt{e}}$. A meta-level evaluation function maps `e` to a value `v`. A self-recognizer `unquote` inverts *quote*, while a self-evaluator `eval` implements *evaluation* on representations. There can be multiple evaluation functions and self-evaluators for a particular language, implementing different evaluation strategies. The thinner arrows indicate mappings up to equivalence: the application of `unquote` to $\overline{\mathtt{e}}$ is *equivalent* to `e`, but is not *identical* to `e`.

There are several examples of self-recognizers with type (1) in the literature. Specifically, Rendel, Ostermann, and Hofer [71] presented the *first* self-recognizer with type (1) for the typed λ-calculus F_ω^*. In previous work we presented self-recognizers with type (1) for System U [19], a typed λ-calculus with *decidable* type checking, and for F_ω [20], a *strongly normalizing* language.

Implementing a self-evaluator with type (2) has remained an open problem until now. Our goal is to identify a core calculus for which we can solve the problem.

The challenge: Can we define a self-evaluator with type (2) for a typed λ-calculus?

Our result: Yes, we present three self-evaluators for a typed λ-calculus with decidable type checking. Our calculus, $F_\omega^{\mu i}$, extends F_ω with recursive types and intensional type functions.

Our starting point is an evaluator for simply-typed λ-calculus (STLC) written in Haskell. The evaluator has type (2) and operates on a representation of STLC based on generalized algebraic data types (GADTs). The gap between the meta-language (Haskell) and the object-language (STLC) is large. To reduce this gap, we apply a series of translations to reduce our GADT-based evaluator of STLC to lower-level constructs: higher-order polymorphism, recursive types, and a theory of type equality. We close the gap in $F_\omega^{\mu i}$, which is designed to support these constructs.

The key challenge of self-representation – "tying the knot" – is to balance the competing needs for a single language to be simultaneously the object language and the meta-language. A more powerful language can represent more, but also has more that needs to be represented. Previous work on self-representation has focused on tying the knot as it pertains to polymorphism [71, 19, 20]. A similar challenge arises for type equality, and this is our main focus in this chapter.

To tie the knot for a language with type equality, we need to consider two questions. First, how expressive must a theory of type equality be in order to implement a typed evaluator for a particular object language? Second, what meta-language features are needed to represent and evaluate a particular theory of type equality? In Section 3.2 we show that to evaluate STLC, type equality between arrow types should be decomposable. In particular, if we know `(A → B) = (S → T)`, then we also know `A = S` and `B = T`. What then is needed to represent and evaluate decomposable type equalities? Haskell implements type equality using built-in type equality coercions [82]. These support decomposition, but have complex typing rules and evaluation semantics that make representation and evaluation difficult. On the other hand, Leibniz equality proofs [68, 90, 9] can be encoded in λ-terms typeable in pure F_ω. This means that representing and evaluating Leibniz equality proofs is no harder than representing and evaluating F_ω. However, Leibniz equality proofs are not decomposable in F_ω. Our goal is to implement a theory of type equality that is decomposable like Haskell's type equality coercions, but that is also easily represented and evaluated, like Leibniz equality proofs.

We achieve our goal by implementing type equality in a new way, by combining Leibniz equality proofs with *intensional type functions* that can depend on the intensional structure of their inputs. The result is an expressive theory of type equality with a simple semantics. This innovation is the key to defining our typed self-representation and self-evaluators.

Our intensional type functions are defined using a `Typecase` operator that is inspired by previous work on intensional type analysis (ITA) [50, 75, 31, 86, 94], but is simpler in three ways:

- We support ITA at the type level only, while previous work supports ITA in types and terms.
- Our `Typecase` operator is not recursive. Previous work used a recursive `Typerec` operator for type-level ITA.
- We support ITA of quantified types without using kind polymorphism.

We present a self-representation of $F_\omega^{\mu i}$ and three self-evaluators with type (2) that operate upon it: one that evaluates terms to weak head normal form, one that performs a single step of left-most reduction, and an implementation of Normalization by Evaluation (NbE) that reduces to β-normal form. The first only reduces closed terms, while the others may reduce under abstractions. We also implement a self-recognizer `unquote` with type (1), and all the benchmark meta-programs from Chapter 2. We have proved that the weak head self-evaluator is correct, and we have implemented and tested our other self-evaluators and meta-programs.

$$\begin{tikzcd} \text{STLC} \arrow[r, "\text{Sections 3 and 4.1}"] & F_\omega^{\mu i} \arrow[loop above, "\text{Sections 4.2, 5, and 6}"] \end{tikzcd}$$

Rest of the chapter. In Section 2 we show how type equality proofs can be used to implement a typed evaluator for STLC in Haskell. In Section 3 we define our calculus $F_\omega^{\mu i}$. In Section 4 we first implement type equality proofs for simple types in $F_\omega^{\mu i}$ and use them to program a typed STLC evaluator. Then we move beyond simple types and extend our type equality proofs to work with quantified and recursive types. In Section 5 we define our self-representation, in Section 6 we present our self-evaluators, in Section 7 we describe our other benchmark meta-programs and our experiments, and in Section 8 we discuss related work.

```
data Exp t where
  Abs :: (Exp t1 → Exp t2) → Exp (t1 → t2)
  App :: Exp (t1 → t2) → Exp t1 → Exp t2

eval :: Exp t → Exp t
eval (App e1 e2) =
  let e1' = eval e1 in
  case e1' of
    Abs f → eval (f e2)
    _     → App e1' e2
eval e = e
```

Figure 3.2: A typed representation of STLC using Haskell GADTs

3.2 From GADTs to Type Equality Proofs

In this section, we will show a series of four evaluators for STLC, all written in Haskell. The idea is for each version to use lower-level constructs than the previous ones, and to use constructs with F_ω types as much as possible. Along the way, we will highlight the techniques needed to typecheck the evaluators.

GADTs. Figure 3.2 shows a representation of Simply-Typed λ-Calculus (STLC) terms in Haskell using GADTs. The representation is Higher-Order Abstract Syntax (HOAS), which means that STLC variables are represented as Haskell variables that range over representations, and we use Haskell functions to bind STLC variables. In the `Abs` constructor, the function type `(Exp t1 → Exp t2)` corresponds to a STLC term of type `Exp t2` that includes a free variable of type `Exp t1`.

Also in Figure 3.2 is a meta-circular evaluator with type (2). That type guarantees that `eval` preserves the type of its input – that the result has the same type. It is meta-circular because it implements STLC features using the corresponding features in the meta-language (Haskell). In particular, we use Haskell β-reduction (function application) to implement STLC β-reduction.

The evaluator `eval` implements weak head-normal evaluation. This means that it reduces the left-most β-redex, but does not evaluate under λ-abstractions or in the argument position of applications. If `e = Abs f`, then `e` is already in weak head-normal form, and `eval e = e`.

```
data Exp t =
   forall t1 t2. (t1 → t2) ∼ t ⇒ Abs (Exp t1 → Exp t2)
 | forall t1. App (Exp (t1 → t)) (Exp t1)

eval :: Exp t → Exp t
eval (App e1 e2) =
  let e1' = eval e1 in
  case e1' of
    Abs f → eval (f e2)
    _     → App e1' e2
eval e = e
```

Figure 3.3: STLC using ADTs and equality coercions

If `e = App e1 e2`, we first recursively evaluate `e1`, letting `e1'` be value of `e1`. If `e1'` is an abstraction `Abs f`, then `App e1' e2` is a redex. We reduce it by applying `f` to `e2`, and then we recursively evaluate the result. If `e1'` is not an abstraction, then we return `App e1' e2`.

We now consider how Haskell type checks `eval`. First, the type annotation on `eval` determines that `App e1 e2` has type `Exp t`. According to the type of `App`, `e1` has type `Exp (t1 → t)` and `e2` has type `Exp t1`, for some type `t1`. Since `eval` preserves the type of its argument, `e1'` also has type `Exp (t1 → t)`. If case analysis finds that `e1'` is of the form `Abs f`, then the type of `Abs` tells us that `f` has the type `Exp t1 → Exp t`.

We can see that Haskell's type checker does some nontrivial work to typecheck code with GADTs. Pattern matching `App e1 e2` introduced the existentially quantified type `t1`. When pattern matching determined that `e1'` is of the form `Abs f`, the type checker aligned the types of `e1'` and `f` so that `f` could be applied to `e2`.

ADTs and equality constraints. GADTs can be understood and implemented as a combination of algebraic data types (ADTs) and equality between types. Figure 3.3 reimplements STLC in this style, using ADTs and Haskell's type equality constraints. In this version, the result type of each constructor of `Exp` is implicitly `Exp t`, while in the GADT version the result type of `Abs` is `Exp (t1 → t2)`. The type equality constraint `(t1 → t2) ∼ t` makes up this difference. Haskell implements GADTs using ADTs and equality constraints [82], so the definitions of `Exp t` in Figures 3.2 and 3.3 are effectively the same. In particular, in both

```
refl   :: Eq t t
sym    :: Eq t1 t2 → Eq t2 t1
trans  :: Eq t1 t2 → Eq t2 t3 → Eq t1 t3
eqApp  :: Eq t1 t2 → Eq (f t1) (f t2)
arrL   :: Eq (t1 → t2) (s1 → s2) → Eq t1 s1
arrR   :: Eq (t1 → t2) (s1 → s2) → Eq t2 s2

coerce :: Eq t1 t2 → t1 → t2
```

Figure 3.4: Interface of explicit type equality proofs

versions the constructors `Abs` and `App` have the same types, and the implementation of `eval` is the same.

Haskell's type equality coercions are reflexive, symmetric, and transitive, and support a number of other rules for deriving equalities. The type checker automatically derives new equalities based on existing ones and inserts coercions based on known equalities. This is how it is able to typecheck `eval`. We refer the interested reader to Sulzmann et al. [82].

Explicit type equality proofs. Figure 3.4 defines an explicit theory of type equality that allows us to derive type equalities and perform coercions manually. We can implement the functions in Figure 3.4 using Haskell, as we show in the appendix, or we can implement them in $F_\omega^{\mu i}$, as we show in Section 3.4.

The basic properties of type equality, namely reflexivity, symmetry, and transitivity, are encoded by `refl`, `sym`, and `trans`, respectively. The only way to introduce a new type equality proof is by using `refl`. `eqApp` shows that equal types are equal in any context. For example, given an equality proof of type `Eq t1 t2`, `eqApp` can derive a proof that `Exp t1` is equal to `Exp t2` by instantiating `f` with `Exp`. The operators `arrL` and `arrR` allow type equality proofs about arrow types to be decomposed into proofs about the domain and codomain types, respectively. We have highlighted them to emphasize their importance in type checking `eval` and in motivating the design of $F_\omega^{\mu i}$. Given a proof of `Eq t1 t2`, `coerce` can change the type of a term from `t1` into `t2`.

Given a closed proof `p` of type `Eq t1 t2`, we expect (1) that it is true that `t1` and `t2` are equal types, and (2) that `coerce p e` evaluates to `e` for all `e`. Open proofs include variables

```
data Exp t =
   forall t1 t2. Abs (Eq (t1 → t2) t) (Exp t1 → Exp t2)
 | forall t1. App (Exp (t1 → t)) (Exp t1)

eval :: Exp t → Exp t
eval (App e1 e2) =
  let e1' = eval e1 in
  case e1' of
    Abs eq f →
      let eqL = eqApp (sym (arrL eq))
          eqR = eqApp (arrR eq)
          f' = coerce eqR . f . coerce eqL
      in eval (f' e2)
    _          → App e1' e2
eval e = e
```

Figure 3.5: STLC using explicit type equality proofs

of equality proof type, which can be thought of as type equality hypotheses. Until these hypothesis are discharged, `coerce p e` should not be reducible to `e`.

ADTs and explicit type equality proofs. Figure 3.5 shows a version of `Exp t` and an evaluator that uses ADTs and explicit type equality proofs. The only difference between this definition of `Exp t` and the one in Figure 3.3 is that we have replaced the type equality constraint `(t1 → t2) ∼ t` with a type equality proof of type `Eq (t1 → t2) t`[1], in order to clarify the role of type equality in type checking `eval`.

As before, we know from the type of `eval` that its argument has type `Exp t`, and `e1` has type `Exp (t1 → t)` and `e2` has type `Exp t1`, for some type `t1`. Since `eval` preserves type, `e1'` also has type `Exp (t1 → t)`.

The differences begin with the pattern match on `e1'`. If `e1'` is of the form `Abs eq f`, then there exist types `s1` and `s2` such that `eq` has the type `Eq (s1 → s2) (t1 → t)` and `f` has the type `Exp s1 → Exp s2`. We use `arrL`, `sym`, and `eqApp` (with `f` instantiated with `Exp`) to derive `eqL`, which has the type `Eq (Exp t1) (Exp s1)`. Similarly, we use `arrR` and `eqApp` to derive `eqR` with the type `Eq (Exp s2) (Exp t)`. Finally, we use coercions based on `eqL` and `eqR` to cast `f` from the type `Exp s1 → Exp s2` to the type

[1]Not to be confused with the type class `Eq` defined in Haskell's Prelude

```
newtype Exp t = Exp {
  matchExp ::
    forall r.
    (forall a b. Eq t (a → b) → (Exp a → Exp b) → r) →
    (forall s. Exp (s → t) → Exp s → r) →
    r
  }

abs :: (Exp t1 → Exp t2) → Exp (t1 → t2)
abs f = Exp (\fAbs fApp → fAbs refl f)

app :: Exp (t1 → t2) → Exp t1 → Exp t2
app e1 e2 = Exp (\fAbs fApp → fApp e1 e2)

eval :: Exp t → Exp t
eval e =
 matchExp e
  (\_ _ → e)
  (\e1 e2 →
   let e1' = eval e1 in
   matchExp e1'
    (\eq f →
     let eqL = eqApp (sym (arrL eq))
         eqR = eqApp (arrR eq)
         f'  = coerce eqR . f . coerce eqL
     in f' e2)
    (\_ _ → app e1' e2))
```

Figure 3.6: Mogensen-Scott encoding of STLC

`Exp t1` → `Exp t`. Thus, `f'` can be applied to `e2`, and its result has type `Exp t`, as required by the type of `eval`.

Mogensen-Scott encoding. By using a typed Mogensen-Scott encoding [63], we can represent STLC using only functions, type equality proofs, and Haskell's `newtype`, a special case of an ADT with only one constructor that has a single field. This version is shown in Figure 3.6. The field of `Exp t` defines a simple pattern-matching interface for STLC representations: given case functions for abstraction and application, each producing a result of type `r`, we can produce an `r`. We manually define constructors `abs` and `app` for `Exp t` by their pattern matching behavior. For example, the arguments to `app` are the two subexpressions of an application node. Given case functions for abstraction and application, `app` calls the case function for application, and passes along its subexpressions. The `abs` constructor is similar, except that it takes one argument, while the case function `fAbs` for abstractions takes two. The first argument to `fAbs` is a type equality proof that `abs` supplies itself.

The function `matchExp` maps representations to their pattern matching interface, and the constructor `Exp` goes the opposite direction. These establish an isomorphism between `Exp t` and its pattern matching interface. In particular, `matchExp (Exp f) = f`. The type `Exp` is recursive because `Exp` occurs in the type of its field `matchExp`.

The Mogensen-Scott encoding of STLC uses higher order (F_ω) types, recursive types, and type equality proofs. In the next section we present $F_\omega^{\mu i}$, which supports each of these features. It extends F_ω with iso-recursive types and intensional type functions that we use to implement the type equality proof interface in Figure 3.4. In Section 4 we define a representation and evaluator for STLC in $F_\omega^{\mu i}$, which are similar to Figure 3.6. Then we go beyond STLC and implement our self-representation and self-evaluator for $F_\omega^{\mu i}$.

3.3 System $F_\omega^{\mu i}$

System $F_\omega^{\mu i}$ is defined in Figure 3.7. It extends F_ω with iso-recursive types and a type operator `Typecase` that is used to define intensional type functions. The kinds are the same as in F_ω. The kind $*$ classifies base types (the types of terms), and arrow kinds classify type level functions. The types are those of F_ω, plus and `Typecase`. The rules of type formation are those of F_ω, plus axioms for and `Typecase`. The terms are those of F_ω, plus `fold` and `unfold` that respectively contract or expand a recursive type. The rules of term formation are those of F_ω, plus rules for `fold` and `unfold`. Notably, there are no new terms that are type checked by `Typecase`. This is different than in previous work on intensional type analysis (ITA), where a type-level ITA operator is used to typecheck a term-level ITA operator. Type equivalence is the same as for F_ω, plus the three reduction rules for `Typecase`. The semantics is full F_ω β-reduction, plus a congruence rule for each of `fold` and `unfold` and a reduction rule for `unfold` combined with `fold`. The normal form terms are those that

$$
\begin{array}{rlllllll}
\text{(kinds)} & \mathsf{K} ::= & * & \mid \mathsf{K}_1 \to \mathsf{K}_2 & & & & \\
\text{(types)} & \mathsf{T} ::= & \mathsf{X} & \mid \mathsf{T}_1 \to \mathsf{T}_2 & \mid \forall \mathsf{X{:}K.T} & \mid \lambda \mathsf{X{:}K.T} \mid \mathsf{T}_1\ \mathsf{T}_2 & \mid \mu & \mid \mathsf{Typecase} \\
\text{(terms)} & \mathsf{e} ::= & \mathsf{x} & \mid \lambda \mathsf{x{:}T.e} & \mid \mathsf{e}_1\ \mathsf{e}_2 & \mid \Lambda \mathsf{X{:}K.e} \mid \mathsf{e\ T} & \mid \mathsf{fold}\ \mathsf{T}_1\ \mathsf{T}_2\ \mathsf{e} & \mid \mathsf{unfold}\ \mathsf{T}_1\ \mathsf{T}_2\ \mathsf{e} \\
\text{(environments)} & \Gamma ::= & \langle\rangle & \mid \Gamma,(\mathsf{x{:}T}) & \mid \Gamma,(\mathsf{X{:}K}) & & &
\end{array}
$$

$$
\begin{array}{rlllll}
\text{(normal form terms)} & \mathsf{v} ::= \mathsf{n} & \mid (\lambda \mathsf{x{:}T.v}) & \mid (\Lambda \mathsf{X{:}K.v}) & \mid \mathsf{fold}\ \mathsf{T}_1\ \mathsf{T}_2\ \mathsf{v} \\
\text{(neutral terms)} & \mathsf{n} ::= \mathsf{x} & \mid \mathsf{n\ v} & \mid \mathsf{n\ T} & \mid \mathsf{unfold}\ \mathsf{T}_1\ \mathsf{T}_2\ \mathsf{n}
\end{array}
$$

Grammar

$$\frac{(\mathsf{X{:}K}) \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{K}}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 : * \qquad \Gamma \vdash \mathsf{T}_2 : *}{\Gamma \vdash \mathsf{T}_1 \to \mathsf{T}_2 : *} \qquad \frac{\Gamma,(\mathsf{X{:}K}) \vdash \mathsf{T} : *}{\Gamma \vdash (\forall \mathsf{X{:}K.T}) : *}$$

$$\frac{\Gamma,(\mathsf{X{:}K}_1) \vdash \mathsf{T} : \mathsf{K}_2}{\Gamma \vdash (\lambda \mathsf{X{:}K}_1\mathsf{.T}) : \mathsf{K}_1 \to \mathsf{K}_2} \qquad \frac{\Gamma \vdash \mathsf{T}_1 : \mathsf{K}_2 \to \mathsf{K} \qquad \Gamma \vdash \mathsf{T}_2 : \mathsf{K}_2}{\Gamma \vdash \mathsf{T}_1\ \mathsf{T}_2 : \mathsf{K}}$$

$$\Gamma \vdash \mu : ((* \to *) \to * \to *) \to * \to *$$

$$
\begin{aligned}
\Gamma \vdash \mathsf{Typecase} : {} & (* \to * \to *) \to \\
& (* \to *) \to (* \to *) \to \\
& (((* \to *) \to * \to *) \to * \to *) \to \\
& *
\end{aligned}
$$

Type Formation

$$\mathsf{T} \equiv \mathsf{T} \qquad \frac{\mathsf{T1} \equiv \mathsf{T2}}{\mathsf{T2} \equiv \mathsf{T1}} \qquad \frac{\mathsf{T1} \equiv \mathsf{T2} \qquad \mathsf{T2} \equiv \mathsf{T3}}{\mathsf{T1} \equiv \mathsf{T3}}$$

$$\frac{\mathsf{T1} \equiv \mathsf{T1}' \qquad \mathsf{T2} \equiv \mathsf{T2}'}{\mathsf{T1} \to \mathsf{T2} \equiv \mathsf{T1}' \to \mathsf{T2}'} \qquad \frac{\mathsf{T} \equiv \mathsf{T}'}{(\forall \mathsf{X{:}K.T}) \equiv (\forall \mathsf{X{:}K.T}')}$$

$$\frac{\mathsf{T} \equiv \mathsf{T}'}{(\lambda \mathsf{X{:}K.T}) \equiv (\lambda \mathsf{X{:}K.T}')} \qquad \frac{\mathsf{T1} \equiv \mathsf{T1}' \qquad \mathsf{T2} \equiv \mathsf{T2}'}{\mathsf{T1\ T2} \equiv \mathsf{T1}'\ \mathsf{T2}'}$$

$$
\begin{aligned}
(\lambda \mathsf{X{:}K.T1})\ \mathsf{T2} &\equiv (\mathsf{T1}[\mathsf{X} := \mathsf{T2}]) \\
(\forall \mathsf{X{:}K.T2}) &\equiv (\forall \mathsf{X}'\mathsf{{:}K.T2}[\mathsf{X} := \mathsf{X}']) \\
(\lambda \mathsf{X{:}K.T}) &\equiv (\lambda \mathsf{X}'\mathsf{{:}K.T}[\mathsf{X} := \mathsf{X}'])
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Typecase\ F1\ F2\ F3\ F4\ (T1 \to T2)} &\equiv \mathsf{F1\ T1\ T2} \\
\mathsf{Typecase\ F1\ F2\ F3\ F4\ (\mu\ T1\ T2)} &\equiv \mathsf{F4\ T1\ T2}
\end{aligned}
$$

$$\frac{\mathsf{X} \notin \mathsf{FV(F3)}}{\mathsf{Typecase\ F1\ F2\ F3\ F4\ (\forall X{:}K.T)} \equiv \mathsf{F2\ (\forall X{:}K.\ F3\ T)}}$$

Type Equivalence

$$\frac{(\mathsf{x{:}T}) \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 : * \qquad \Gamma,(\mathsf{x{:}T}_1) \vdash \mathsf{e} : \mathsf{T}_2}{\Gamma \vdash (\lambda \mathsf{x{:}T}_1\mathsf{.e}) : \mathsf{T}_1 \to \mathsf{T}_2}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_2 \to \mathsf{T} \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \mathsf{T}}$$

$$\frac{\Gamma,(\mathsf{X{:}K}) \vdash \mathsf{e} : \mathsf{T}}{\Gamma \vdash (\Lambda \mathsf{X{:}K.e}) : (\forall \mathsf{X{:}K.T})}$$

$$\frac{\Gamma \vdash \mathsf{e} : (\forall \mathsf{X{:}K.T}_1) \qquad \Gamma \vdash \mathsf{T}_2 : \mathsf{K}}{\Gamma \vdash \mathsf{e} : \mathsf{T}_1[\mathsf{X{:=}T}_2]}$$

$$\frac{\Gamma \vdash \mathsf{F} : (* \to *) \to * \to * \qquad \Gamma \vdash \mathsf{T} : * \qquad \Gamma \vdash \mathsf{e} : \mathsf{F}\ (\mu\ \mathsf{F})\ \mathsf{T}}{\Gamma \vdash \mathsf{fold\ F\ T\ e} : \mu\ \mathsf{F\ T}}$$

$$\frac{\Gamma \vdash \mathsf{F} : (* \to *) \to * \to * \qquad \Gamma \vdash \mathsf{T} : * \qquad \Gamma \vdash \mathsf{e} : \mu\ \mathsf{F\ T}}{\Gamma \vdash \mathsf{unfold\ F\ T\ e} : \mathsf{F}\ (\mu\ \mathsf{F})\ \mathsf{T}}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{T}_1 \qquad \mathsf{T}_1 \equiv \mathsf{T}_2 \qquad \Gamma \vdash \mathsf{T}_2 : *}{\Gamma \vdash \mathsf{e} : \mathsf{T}_2}$$

Term Formation

$$
\begin{aligned}
(\lambda \mathsf{x{:}T.e})\ \mathsf{e}_1 &\longrightarrow \mathsf{e}[\mathsf{x} := \mathsf{e}_1] \\
(\Lambda \mathsf{X{:}K.e})\ \mathsf{T} &\longrightarrow \mathsf{e}[\mathsf{X} := \mathsf{T}] \\
\mathsf{unfold\ F\ T\ (fold\ F'\ T'\ e)} &\longrightarrow \mathsf{e}
\end{aligned}
$$

$$\frac{\mathsf{e}_1 \longrightarrow \mathsf{e}_2}{\begin{aligned}
\mathsf{e}_1\ \mathsf{e}_3 &\longrightarrow \mathsf{e}_2\ \mathsf{e}_3 \\
\mathsf{e}_3\ \mathsf{e}_1 &\longrightarrow \mathsf{e}_3\ \mathsf{e}_2 \\
\mathsf{e}_1\ \mathsf{T} &\longrightarrow \mathsf{e}_2\ \mathsf{T} \\
(\lambda \mathsf{x{:}T.e}_1) &\longrightarrow (\lambda \mathsf{x{:}T.e}_2) \\
(\Lambda \mathsf{X{:}K.e}_1) &\longrightarrow (\Lambda \mathsf{X{:}K.e}_2) \\
\mathsf{fold\ F\ T\ e}_1 &\longrightarrow \mathsf{fold\ F\ T\ e}_2 \\
\mathsf{unfold\ F\ T\ e}_1 &\longrightarrow \mathsf{unfold\ F\ T\ e}_2
\end{aligned}}$$

Reduction

Figure 3.7: Definition of $F_\omega^{\mu i}$

cannot be reduced. Following Girard et al. [44], we define normal forms simultaneously with neutral terms, which are the normal forms other than abstractions or `fold`. Intuitively, a neutral term can replace a variable in a normal form without introducing a redex.

Capital letters and capitalized words such as `F`, `Exp`, `Bool` range over types. We will often use `F` for higher-kinded types (type functions), and `A`, `B`, `S`, `T`, `X`, `Y` for type variables of kind $*$. Lower case letters and uncapitalized words range over terms.

Recursive types can be used to define recursive functions and data types defined in terms of themselves. For example, each of the three versions of `Exp` defined in Figures 3.2, 3.3, and 3.6 is recursive. An iso-recursive type is not equal (or equivalent) to its definition, but rather is isomorphic to it, and `fold` and `unfold` form the isomorphism: `unfold` maps a recursive type to its definition, and `fold` is the inverse. Intuitively, `fold` generalizes the `Exp` newtype constructor from Figure 3.6 to work for many data types. Similarly, `unfold` generalizes `matchExp`. Using iso-recursive types is important for making type checking decidable. For more information about iso-recursive types, we refer the interested reader to Pierce's book[70].

To simplify the language and our self-representation, we only support recursive types of kind $* \rightarrow *$ (type functions). This is sufficient for our needs, which are to encode recursive data types in the style seen in the previous section, and to define recursive functions. We can encode recursive base types (types of kind $*$) using a constant type function.

We will discuss `Typecase` in detail in Section 3.3.3.

3.3.1 Metatheory

System $F_\omega^{\mu i}$ is type safe and type checking is decidable. Proofs are included in the appendix. For type safety, we use a standard Progress and Preservation proof [97]. For decidability of type checking, we show that reduction of types is confluent and strongly normalizing [64].

Theorem 3.3.1. *[Type Safety]*
If $\langle\rangle \vdash \mathsf{e} : \mathsf{T}$, *then either* e *is a normal form, or there exists an* e' *such that* $\langle\rangle \vdash \mathsf{e}' : \mathsf{T}$ *and* $\mathsf{e} \longrightarrow \mathsf{e}'$.

Theorem 3.3.2. *Type checking is decidable.*

3.3.2 Syntactic Sugar and Abbreviations

System $F_\omega^{\mu i}$ is a low-level calculus, more suitable for theory than for real-world programming. We use the following syntactic sugar to make our code more readable. We highlight the syntactic sugar to distinguish it from the core language.

- `let x : T = e1 in e2` desugars to`(λx:T.e2) e1`, as usual.
- `let rec x : T1 = e1 in e2` desugars to `let x : T1 = fix T1 (λx:T1. e1) in e2`. Here `fix` is a standard fixpoint combinator of type `∀T:*. (T → T) → T`.
- `decl X : K = T;` defines a new type abbreviation. `T` is inlined at every subsequent occurrence of `X`. Similarly, `decl x : T = e;` defines an inlined term abbreviation.
- `decl rec x : T = e;` declares a recursive term. It uses `fix` like `let rec`, and inlines like `decl`.

For further brevity, we sometimes omit the type annotations on abstractions, let bindings or declarations, when the type can be easily inferred from context. For example, we will write `(λx.e)` instead of `(λx:T.e)`. We use `f ∘ g` to denote the composition of (type or term) functions `f` and `g`. This desugars to `(λx. f (g x))`, where `x` is fresh.

We use `S × T` for pair types, which can be easily encoded in System $F_\omega^{\mu i}$. Intuitively, `×` is an infix type function of kind $* \to * \to *$. We use `(x,y)` to construct the pair of `x` and `y`. `fst` and `snd` project the first and second component from a pair, respectively.

3.3.3 Intensional Type Functions

Our `Typecase` operator allows us to define type functions that depend on the top-level structure of a base type. It is parameterized by four case functions, one for arrow types, two for quantified types, and one for recursive types. When applied to an arrow type or a recursive

```
decl ⊥ : * = (∀T:*. T);
decl ArrL : ∗ → ∗ =
 Typecase (λA:∗. λB:∗. A) (λA:∗.⊥) (λA:∗.⊥)
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. ⊥);
decl ArrR : ∗ → ∗ =
 Typecase (λA:∗. λB:∗. B) (λA:∗.⊥) (λA:∗.⊥)
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. ⊥);
decl All : (∗ → ∗) → (∗ → ∗) → ∗ → ∗ =
 λOut:∗ → ∗. λIn:∗ → ∗.
 Typecase (λA:∗. λB:∗. B) Out In
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. ⊥);
decl Unfold : ∗ → ∗ =
 Typecase (λA:∗. λB:∗. ⊥) (λA:∗.⊥) (λA:∗.⊥)
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. F (μ F) A);
```

Figure 3.8: Intensional type functions

type, `Typecase` decomposes the input type and applies the corresponding case function to the components. For example, when applied to an arrow type `T1` → `T2`, `Typecase` applies the case function for arrows to `T1` and `T2`. When applied to a recursive type `μ F T`, `Typecase` applies the case function for recursive types to `F` and `T`.

We have two functions for the case of quantified types because they cannot be easily decomposed in $F_\omega^{\mu i}$. Previous work on ITA for quantified types [86] would decompose a quantified type ∀`X:K.T` into the kind `K` and a type function of kind `K` → ∗. The components would then be passed as arguments to a case function for quantified types. This approach requires kind polymorphism in types, which is outside of $F_\omega^{\mu i}$. Our solution uses two functions for quantified types. `Typecase` applies one function inside the quantified type (under the quantifier), and the other outside.

For example, let `F` be an intensional type function defined by `F = Typecase Arr Out In Mu`. Here, `Arr` and `Mu` are the case functions for arrow types and recursive types, respectively. `Out` and `In` are the case functions for quantified types, with `Out` being applied outside the type, and `In` inside the type. Then `F (∀X:K.T)` ≡ `Out (∀X:K. In T)`. Note that to avoid variable capture, we require that `X` not occur free in `In` (which can be ensured by renaming `X`).

Figure 3.8 defines four intensional type functions. Each expects its input type to be of a particular form: `ArrL` and `ArrR` expect an arrow type, `All` expects a quantified type, and

`Unfold` expects a recursive type. On types not of the expected form, each function returns the type $\bot$=`(`$\forall$`T:*.T)`, which we use to indicate an error. The type $\bot$ is only inhabited by non-normalizing terms.

`ArrL` and `ArrR` return the domain and codomain of an arrow type, respectively. More precisely, the specification of `ArrL` is as follows (`ArrR` is similar):

$$\texttt{ArrL T} \equiv \begin{cases} \texttt{T1} & \text{if} \quad \texttt{T} \equiv \texttt{T1} \rightarrow \texttt{T2} \\ \bot & \text{otherwise} \end{cases}$$

`All` takes two type functions `Out` and `In`, and applies them outside and inside a $\forall$ quantifier, respectively.

$$\texttt{All Out In T} \equiv \begin{cases} \texttt{Out } (\forall\texttt{X:K. In T}) & \text{if} \quad \texttt{T} \equiv \forall\texttt{X:K. T} \\ \bot & \text{otherwise} \end{cases}$$

`Unfold` returns the result of unfolding a recursive type one time:

$$\texttt{Unfold T} \equiv \begin{cases} \texttt{F (}\mu\texttt{ F) A} & \text{if} \quad \texttt{T} \equiv \mu\texttt{ F A} \\ \bot & \text{otherwise} \end{cases}$$

In the next section, we will use these intensional type functions to define type equality proofs that are useful for defining GADT-style typed representations and polymorphically-typed self evaluators.

3.4 Type Equality Proofs in $F_\omega^{\mu i}$

In Section 3.4.1 we implement decomposable type equality proofs in $F_\omega^{\mu i}$ and use them to represent and evaluate STLC. Then in Section 3.4.2 we go beyond simple types to quantified and recursive types in preparation for our $F_\omega^{\mu i}$ self-representation and self-evaluators.

```
decl Eq : ∗ → ∗ =
 λA:∗. λB:∗. ∀F:∗ → ∗. F A → F B;

decl refl : (∀A:∗. Eq A A) =
 ΛA:∗. ΛF:∗ → ∗. λx : F A. x;

decl sym : (∀A:∗. ∀B:∗. Eq A B → Eq B A) =
 ΛA:∗. ΛB:∗. λeq : Eq A B.
 let p : Eq A A = refl A in
 eq (ΛT:∗. Eq T A) p;

decl trans : (∀A:∗. ∀B:∗. ∀C:∗.
              Eq A B → Eq B C → Eq A C) =
 ΛA:∗. ΛB:∗. ΛC:∗. λeqAB:Eq A B. λeqBC:Eq B C.
 ΛF:∗ → ∗. λx:F A. eqBC F (eqAB F x);

decl eqApp : (∀A:∗. ∀B:∗. ∀F:∗ → ∗.
            Eq A B → Eq (F A) (F B)) =
  ΛA:∗. ΛB:∗. ΛF:∗ → ∗. λeq : Eq A B.
  let p : Eq (F A) (F A) = refl (F A) in
  eq (λT:∗. Eq (F A) (F T)) p;

decl arrL : (∀A1:∗. ∀A2:∗. ∀B1:∗. ∀B2:∗.
             Eq (A1 → A2) (B1 → B2) →
             Eq A1 B1) =
 ΛA1 A2 B1 B2. eqApp (A1→A2) (B1→B2) ArrL;

decl arrR : (∀A1:∗. ∀A2:∗. ∀B1:∗. ∀B2:∗.
             Eq (A1 → A2) (B1 → B2) →
             Eq A2 B2) =
 ΛA1 A2 B1 B2. eqApp (A1→A2) (B1→B2) ArrR;

decl coerce : (∀A:∗. ∀B:∗. Eq A B → A → B) =
 ΛA:∗. ΛB:∗. λeq:Eq A B. eq Id;
```

Figure 3.9: Implementation of type equality proofs in $F_\omega^{\mu i}$.

3.4.1 Equality Proofs for Simple Types

Figure 3.9 shows the $F_{\omega}^{\mu i}$ implementation of the type equality proofs from Figure 3.4. The foundation of our encoding is Leibniz equality, which encodes that two types are indistinguishable in all contexts. This is a standard technique for encoding type equality in F_{ω} [68, 90, 9]. The type `Eq A B` is defined as `∀F:∗ → ∗. F A → F B`. Intuitively, the type function `F` ranges over type contexts, and a Leibniz equality proof can replace the type `A` with `B` in any context `F`.

The only way to introduce a new type equality proof is by `refl`, which constructs an identity function to witness that a type is equal to itself. Symmetry is encoded by `sym`, which uses an equality proof of type `Eq A B` to coerce another proof of type `Eq A A`, replacing the first `A` with `B` and resulting in the type `Eq B A`. Transitivity is encoded by `trans`, which uses function composition to combine two coercions. A proof of type `Eq A B` is effectively a coercion – it can coerce any term of type `F A` to `F B`. Thus, `coerce` simply instantiates the proof with the identity function on types. For brevity we will sometimes omit `coerce` and use equality proofs as coercions directly.

Each of `Eq`, `refl`, `sym`, `trans`, `eqApp`, and `coerce` are definable in the pure F_{ω} subset of $F_{\omega}^{\mu i}$. The addition of intensional type functions allows $F_{\omega}^{\mu i}$ to decompose Leibniz equality proofs. The key is that `eqApp` is stronger in $F_{\omega}^{\mu i}$ than in F_{ω} because the type function `F` can be intensional. In particular, `arrL` and `arrR` are defined using `eqApp` with the intensional type functions `ArrL` and `ArrR`, respectively.

Figure 3.10 shows the STLC representation and evaluator in $F_{\omega}^{\mu i}$. It uses a Mogensen-Scott encoding similar to the one in Figure 3.6, with a few notable differences. The type `ExpF` is a stratified version of `Exp`. In particular, it uses a λ-abstraction to untie the recursive knot. `Exp` is defined as `μ ExpF`, which re-ties the knot. Now `Exp T` and `ExpF Exp T` are isomorphic, with `unfold ExpF T` converting from `Exp T` to `ExpF Exp T`, and `fold ExpF T` converting from `ExpF Exp T` back to `Exp T`. We define `matchExp` as a convenience and to align with Figure 3.6, but we could as well use `unfold ExpF T` instead of `matchExp T`. This version of `eval` is similar to the previous version. The main difference is in the increased amount of type annotations.

```
decl ExpF : (∗ → ∗) → ∗ → ∗ =
 λExp : ∗ → ∗. λT : ∗. ∀R : ∗.
 (∀A:∗.∀B:∗. Eq (A→B) T → (Exp A → Exp B) → R) →
 (∀S:∗. Exp (S → T) → Exp S → R) →
 R;

decl Exp : ∗ → ∗ = μ ExpF;

decl abs:(∀A:∗.∀B:∗. (Exp A → Exp B) → Exp (A→B)) =
 ΛA:∗. ΛB:∗. λf:Exp A → Exp B.
 fold ExpF (A → B)
  (ΛR. λfAbs. λfApp. fAbs A B (refl (A → B)) f);

decl app:(∀A:∗.∀B:∗. Exp (A→B) → Exp A → Exp B) =
 ΛA:∗. ΛB:∗. λe1 : Exp (A → B). λe2 : Exp A.
 fold ExpF B (ΛR. λfAbs. λfApp. fApp A e1 e2);

decl matchExp : (∀T:∗. Exp T → ExpF Exp T) =
 ΛT : ∗. λe : Exp T. unfold ExpF T e;

decl rec eval : (∀T:∗. Exp T → Exp T) =
 ΛT:∗. λe : Exp T.
 matchExp T e (Exp T)
  (ΛT1 T2. λeq f. e)
  (ΛS:∗. λe1 : Exp (S → T). λe2 : Exp S.
   let e1':Exp (S → T) = eval (S → T) e1 in
   matchExp (S → T) e1' (Exp T)
    (ΛA B. λeq:Eq (A→B) (S→T). λf:Exp A → Exp B.
     let eqL:Eq (Exp S) (Exp A) =
      eqApp S A Exp (sym A S (arrL A B S T eq)) in
     let eqR:Eq (Exp B) (Exp T) =
      eqApp B T Exp (arrR A B S T eq) in
     let f':Exp S → Exp T = λx : Exp S.
      let x' : Exp A = coerce (Exp S) (Exp A) eqL x in
      coerce (Exp B) (Exp T) eqR (f x')
     in
     eval T (f' e2))
    (ΛT2. λe3 e4. app S T e1' e2));
```

Figure 3.10: Encoding and evaluation of STLC in $F_\omega^{\mu i}$

```
decl TcAll : * → * =
 λT.∀Arr.∀Out.∀In.∀Mu.
 Eq (Typecase Arr Out In Mu T) (All Out In T);
decl UnAll : * → * =
 λT.∀Out. Eq (All Out (λA:*.A) T) (Out T);
decl IsAll : * → * = λT. (TcAll T × UnAll T);
```

$\texttt{tcAll}_{\texttt{X,K,T}}$ = ΛArr.ΛOut.ΛIn.ΛMu.refl (Out(∀X:K.In T))
$\texttt{unAll}_{\texttt{X,K,T}}$ = ΛOut.refl (Out (∀X:K.T))
$\texttt{isAll}_{\texttt{X,K,T}}$ = ($\texttt{tcAll}_{\texttt{X,K,T}}$, $\texttt{unAll}_{\texttt{X,K,T}}$)

Figure 3.11: IsAll proofs.

3.4.2 Beyond Simple Types

In this section, we move beyond STLC in preparation for our self-representation of $F_\omega^{\mu i}$. We will focus on the question: How can we establish that an unknown type is a quantified or recursive type?

In Figure 3.10, we establish that a type `T` is an arrow type by abstracting over types `T1` and `T2` of kind $*$ and a proof of type `Eq (T1 → T2) T`. This will work for any arrow type because `T1` and `T2` must have kind $*$ in order for `T1 → T2` to kind check. The case for recursive types is similar. In $F_\omega^{\mu i}$, a recursive type `μ F A` kind checks only if `F` has kind `(* → *) → * → *` and `A` has kind $*$. Therefore, we can establish that some type `T` is a recursive type by abstracting over `F` and `A` and a proof of type `Eq (μ F A) T`. Given a proof that one recursive type `μ F1 A1` is equal to another `μ F2 A2`, we know that their unfoldings are equal as well. This can be proved using `eqApp` and the intensional type function `Unfold`:

```
decl eqUnfold : (∀F1:(* → *) → * → *. ∀A1:*.
                 ∀F2:(* → *) → * → *. ∀A2:*.
                 Eq (μ F1 A1) (μ F2 A2) →
                 Eq (F1 (μ F1) A1) (F2 (μ F2) A2) =
  ΛF1 A1 F2 A2. eqApp (μ F1 A1) (μ F2 A2) Unfold;
```

We establish that a type is a quantified type in a different way, by proving equalities about the behavior of `Typecase` on that type. We do this because unlike arrow types and recursive types, we can't abstract over the components of an arbitrary quantified type in $F_\omega^{\mu i}$, as was discussed earlier in Section 3.3.3. Figure 3.11 defines our `IsAll` proofs that a type is a quantified type. A proof of type `IsAll T` consists of a pair of *polymorphic* equality proofs about `Typecase`. The first is of type `TcAll T` and proves that because `T` is a quantified type, `Typecase Arr Out In Mu T` is equal to `All Out In T`. The proof is polymorphic because it proves the equality for any `Arr` and `Mu`. In other words, `Arr` and `Mu` are irrelevant: since `T` is a quantified type, they can be replaced by the constant $\perp$ functions used by `All`. The second polymorphic equality proof, of type `UnAll T`, shows that `All Out (λA:∗.A) T` is equal to `Out T` for any `Out`. This is true because applying the identity function under the quantifier has no effect. These proofs are orthogonal to each other, and each is useful for some of our operations, as we discuss in Section 3.7.

We define `IsAll` proofs using indexed abbreviations $\texttt{tcAll}_{\texttt{X,K,T}}$, $\texttt{unAll}_{\texttt{X,K,T}}$, and $\texttt{isAll}_{\texttt{X,K,T}}$. These are meta-level abbreviations, not part of $F_\omega^{\mu i}$. The type of $\texttt{tcAll}_{\texttt{X,K,T}}$ is `TcAll (∀X:K.T)`, the type of $\texttt{unAll}_{\texttt{X,K,T}}$ is `UnAll (∀X:K.T)`, and the type of $\texttt{isAll}_{\texttt{X,K,T}}$ is `IsAll (∀X:K.T)`. Note that $\texttt{tcAll}_{\texttt{X,K,T}}$ and $\texttt{unAll}_{\texttt{X,K,T}}$ use `refl` to create the equality proof. In the case of $\texttt{unAll}_{\texttt{X,K,T}}$, the proof `refl (Out (∀X:K.T)` has type `Eq (Out (∀X:K.T)) (Out (∀X:K.T))`, which is equivalent to the type `Eq (All Out (λA:∗.A) (∀X:K.T)) (Out (∀X:K.T))`.

Impossible Cases. It is sometimes impossible to establish that a type is of a particular form, in particular, if it is already known to be of a different form. This sometimes happens when pattern matching on a GADT. For example, suppose we added integers to our Haskell representation of STLC. When matching on a representation of type `Exp Int`, the `Abs` case would provide a proof that `Int` is equal to an arrow type `t1 → t2`, which is impossible. Haskell's type checker can detect that such cases are unreachable, and therefore those cases need not be covered in order for a pattern match expression to be exhaustive.

Our equality proofs support similar reasoning about impossible cases, which we use in some of our meta-programs. In particular, given an impossible type equality proof (which must be hypothetical), we can derive a (strongly normalizing) term of type $\perp$:

```
eqArrMu  : ∀A B F T. Eq (A → B) (μ F T) → ⊥
arrIsAll : ∀A B. IsAll (A → B) → ⊥
muIsAll  : ∀F T. IsAll (μ F T) → ⊥
```

There are three kinds of contradictory equality proofs in $F_\omega^{\mu i}$: a proof that an arrow type is equal to a recursive type (`eqArrMu`), that an arrow type is a quantified type (`arrIsAll`), or that a recursive type is a quantified type(`muIsAll`). Definitions of `eqArrMu`, `arrIsAll`, and `muIsAll` are provided in the appendix.

3.5 Our Representation Scheme

The self-representation of System $F_\omega^{\mu i}$ is shown in Figure 3.12. Like the STLC representation in Figure 3.10, we use a typed Mogensen-Scott encoding, though there are several important differences. Following previous work on typed self-representation [71, 19, 20], we use Parametric Higher-Order Abstract Syntax (PHOAS) [29, 92] to give our representation more expressiveness. The type `PExp` is parametric in `V`, which determines the type of free variables in a representation. Intuitively, `PExp` can be understood as the type of representations that may contain free variables. The type `Exp` quantifies over `V`, which ensures that the representation is closed. Our quoter assumes that the designated variable `V` is globally fresh.

Our quotation procedure is similar to previous work on typed self-representation [71, 19, 20]. The quotation function $\overline{\cdot}$ is defined only on closed terms, and depends on a pre-quotation function $\triangleright$ from type derivations to terms. In the judgment $\Gamma \vdash$ `e : T` $\triangleright$ `q`, the input is the type derivation $\Gamma \vdash$ `e : T`, and the output is a term `q`. We call `q` the pre-representation of `e`.

We represent variables meta-circularly, that is, by other variables. In particular, a variable of type `T` is represented by a variable of the same name with type `PExp V T`. The cases for quoting λ-abstraction, application, fold and unfold are similar. In each case, we recursively quote the subterm and apply the corresponding constructor. The constructors for these cases create the necessary type equality proofs.

To represent type abstraction and application, the quoter generates `IsAll` proofs itself, since they depend on the kind of the type (which cannot be passed as an argument to the constructors in $F_\omega^{\mu i}$). The quoter also generates utility functions $\texttt{stripAll}_{\texttt{K}}$, $\texttt{underAll}_{\texttt{X,K,T}}$, and $\texttt{inst}_{\texttt{X,K,T,S}}$ that are useful to meta-programs for operating on type abstractions and applications. These utility functions comprise an extensional representation of polymorphism that is similar to one we developed for F_ω in previous work [20]. The purpose of the extensional representation is to represent polymorphic terms in languages like F_ω and $F_\omega^{\mu i}$ that do not include kind polymorphism.

The function $\texttt{stripAll}_{\texttt{K}}$ has the type `StripAll (∀X:K.T)`, as long as `(∀X:K.T)` is well-typed. For any type `A` in which `X` does not occur free, $\texttt{stripAll}_{\texttt{K}}$ can map `All Id (λB:*.A) (∀X:K.T)` ≡ `(∀X:K.A)` to `A`. Note that the quantification of `X` is redundant, since it does not occur free in the type `A`. Therefore, any instantiation of `X` will result in `A`. We use the fact that all kinds in $F_\omega^{\mu i}$ are inhabited to define $\texttt{stripAll}_{\texttt{K}}$. It uses the kind inhabitant $\texttt{T}_{\texttt{K}}$ for the instantiation. For each kind `K`, $\texttt{T}_{\texttt{K}}$ is a closed type of kind `K`.

The function $\texttt{underAll}_{\texttt{X,K,T}}$ has the type `UnderAll (∀X:K.T)`. It can apply a function under the quantifier of a type `All Id F1 (∀X:K.T)` ≡ `(∀X:K. F1 T)` to produce a result of type `All Id F2 (∀X:K.T)` ≡ `(∀X:K. F2 T)`. In particular, our evaluators use
$\texttt{underAll}_{\texttt{X,K,T}}$ to make recursive calls on the body of a type abstraction. The representation of a type abstraction `(ΛX:K.e)` of type `(∀X:K.T)` contains the term `(ΛX:K.q)`. Here, `q` is the representation of `e`, in which the type variable `X` can occur free. The type of `(ΛX:K.q)` is `All Id (PExp V) (∀X:K.T)` ≡ `(∀X:K. PExp V T)`.

We can use $\texttt{underAll}_{\texttt{X,K,T}}$ and $\texttt{stripAll}_{\texttt{K}}$ together in operations that always produce results of a particular type. For example, our measure of the size of a representation always returns a `Nat`. We use $\texttt{underAll}_{\texttt{X,K,T}}$ to make the recursive call to `size` under the quantifier. The result of $\texttt{underAll}_{\texttt{X,K,T}}$ has the type `All Id (λY:*. Nat) (∀X:K.T)` ≡ `(∀X:K.Nat)` where the quantification of `X` is redundant. We can then use $\texttt{strip}_{\texttt{K}}$ to strip away the redundant quantifier, enabling us to access the `Nat` underneath.

An instantiation function $\texttt{inst}_{\texttt{X,K,T,S}}$ has the type `Inst (∀X:K.T) (T[X:=S])`. It can be used to instantiate types of the form `(∀X:K. F T)`, producing instantiations of the form `F (T[X:=S])`.

The combination of `IsAll` proofs and the utility functions
$\texttt{stripAll}_{\texttt{K}}$, $\texttt{underAll}_{\texttt{X,K,T}}$, and $\texttt{inst}_{\texttt{X,K,T,S}}$ allows us to represent higher-kinded polymorphism without kind polymorphism. Notice that in the types of `tabs` and `tapp` (shown in Figure

3.12), the type variables `A` range over arbitrary quantified types. The `IsAll` proofs and utility functions witness that `A` is a quantified type and provide an interface for working on quantified types that is expressive enough to support a variety of meta-programs.

Properties. Every closed and well-typed term has a unique representation that is also closed and well-typed.

Theorem 3.5.1. *If* $\langle\rangle \vdash$ `e : T`*, then* $\langle\rangle \vdash \overline{\texttt{e}}$ `: Exp T`*.*

The proof is by induction on the derivation of the typing judgment $\langle\rangle \vdash$ `e : T`. It relies on the fact that we can always produce the proof terms and utility functions needed for each constructor.

All representations are strongly normalizing, even those that represent non-normalizing terms.

Theorem 3.5.2. *If* $\langle\rangle \vdash$ `e : T`*, then* $\overline{\texttt{e}}$ *is strongly normalizing.*

3.6 Our Self-Evaluators

In this section we discuss our three self-evaluators, which implement weak head normal form evaluation, single-step left-most β-reduction, and normalization by evaluation (NbE).

Weak head-normal evaluation. Figure 3.13 shows our first evaluator, which reduces terms to their weak head-normal form. The closed and well-typed weak head-normal forms of $\mathrm{F}_\omega^{\mu i}$ are λ and Λ abstractions, and `fold` expressions. There is no evaluation under abstractions or `fold` expressions, and function arguments are not evaluated before β-reduction.

The function `eval` evaluates closed representations, which have `Exp` types. The main evaluator is `evalV`, which operates on `PExp` types. If the input is a variable, a λ or Λ abstraction, or a `fold` expression, it is already a weak head-normal form. We use constant case functions `constVar`, `constAbs`, etc. to return the input in these cases. The case for application is similar to that for STLC from Figure 3.10, except for the use of the utility

$\mathtt{T}_{*} = (\forall \mathtt{X}{:}*.\mathtt{X})$ $\qquad$ $\mathtt{T}_{\mathtt{K1}\to\mathtt{K2}} = \lambda \mathtt{X}{:}\mathtt{K1}.\mathtt{T}_{\mathtt{K2}}$

Kind Inhabitants

```
decl Id : ∗ → ∗ = λA:∗. A;

decl UnderAll : ∗ → ∗ =
 λT:∗. ∀F1:∗ → ∗. ∀F2:∗ → ∗.
 (∀A:∗. F1 A → F2 A) →
 All Id F1 A → All Id F2 A;
decl StripAll : ∗ → ∗ =
  λT:∗. ∀A:∗. All Id (λB:∗. A) T → A;
decl Inst : ∗ → ∗ → ∗ =
  λA:∗. λB:∗. ∀F:∗→∗. All Id F A → F B;

underAll_{X,K,T} =
 ΛF1. ΛF2. λf : (∀A:∗. F1 A → F2 A).
 λe : (∀X:K. F1 T). ΛX:K. f T (e X)
stripAll_{K} = ΛA. λe:(∀X:K.A). e T_{K}
inst_{X,K,T,S} = ΛF.λf:(∀X:K.F T).f S
```

Operators on quantified types.

```
decl PExpF : (∗ → ∗) → (∗ → ∗) → ∗ → ∗ =
 λV:∗ → ∗. λPExpV:∗ → ∗. λA:∗. ∀R:∗.
 (V A → R) →
 (∀S T. Eq (S→T) A → (PExp V S → PExp V T) → R) →
 (∀B. PExp V (B → A) → PExp V B → R) →
 (IsAll A → StripAll A → UnderAll A →
  All Id (PExp V) A → R) →
 (∀B:∗. IsAll B → Inst B A → PExp V B → R) →
 (∀F B. Eq (μ F B) A → PExp V (F (μ F) B) → R) →
 (∀F B. Eq (F (μ F) B) A → PExp V (μ F B) → R) →
 R;

decl PExp : (∗→∗)→∗→∗ = λV:∗→∗. μ (PExpF V);
decl Exp : ∗ → ∗ = λA:∗. ∀V:∗ → ∗. PExp V A;
```

Definitions of PExp and Exp

```
decl var:(∀V A. V A → PExp V A) =
 ΛV A. λx:V A. fold (PExpF V) A (
 λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
 var x);
decl abs:(∀V A B. (PExp V A → PExp V B) → PExp V (A → B)) =
 ΛV A B.λf:(PExp V A → PExp V B). fold (PExpF V) (A → B) (
 λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
 abs A B (refl (A → B)) f);
decl app:(∀V A B. PExp V (B→A) → PExp V B → PExp V A) =
 ΛV A B.λf:PExp V (B→A).λx:PExp V B. fold (PExpF V) A (
 λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
 app B f x);
decl tabs:(∀V A. IsAll A → StripAll A → UnderAll A →
        All Id (PExp V) A → PExp V A) =
 ΛV A. λp:IsAll A. λs:StripAll A. λu:UnderAll A.
 λe:All Id (PExp V) A. fold (PExpF V) A (
 λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
 tabs p s u e);
decl tapp:(∀V A B.IsAll A→Inst A B→PExp V A→PExp V B)=
 ΛV A B. λp:IsAll A. λi:Inst A B. λe:PExp V A.
 fold (PExpF V) B (
  λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
  tapp A p i e);
decl fld:(∀V F A. PExp V (F (μ F) A) → PExp V (μ F A)) =
 ΛV F A. λe:PExp V (F (μ F) A). fold (PExpF V) (μ F A) (
  λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
  fld F A (refl (μ F A)) e);
decl unfld:(∀V F A. PExp V (μ F A) → PExp V (F (μ F) A)) =
 ΛV F A. λe : PExp V (μ F A). fold (PExpF V) (F (μ F) A) (
  λR.λvar.λabs.λapp.λtabs.λtapp.λfld.λunfld.
  unfld F A (refl (F (μ F) A)) e);

decl matchExp : (∀V A. PExp V A → PExpF (PExp V) A) =
 ΛV:∗→∗.ΛA:∗.λe:PExp V A. unfold (PExpF V) A e;
```

Constructors and match for PExp

$$\frac{(\mathtt{x} : \mathtt{T}) \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \triangleright \mathtt{x}}$$

$$\frac{\Gamma \vdash \mathtt{e} : (\forall \mathtt{X{:}K.T}) \triangleright \mathtt{q} \qquad \mathtt{isAll}_{\mathtt{X,K,T}} = \mathtt{p} \qquad \Gamma \vdash \mathtt{A} : \mathtt{K} \qquad \mathtt{inst}_{\mathtt{X,K,T,A}} = \mathtt{i}}{\Gamma \vdash \mathtt{e\ A} : \mathtt{T[X{:}=A]} \triangleright \mathtt{tapp\ V\ (\forall X{:}K.T)\ (T[X{:}=A])\ p\ i}}$$

$$\frac{\Gamma \vdash \mathtt{T1} : * \qquad \Gamma, (\mathtt{x{:}T1}) \vdash \mathtt{e} : \mathtt{T2} \triangleright \mathtt{q}}{\Gamma \vdash (\lambda \mathtt{x{:}T1.e}) : \mathtt{T1} \to \mathtt{T2} \triangleright \mathtt{abs\ V\ T1\ T2}\ (\lambda \mathtt{x{:}PExp\ V\ T1.\ q})}$$

$$\frac{\Gamma \vdash \mathtt{F} : (* \to *) \to * \to * \qquad \Gamma \vdash \mathtt{T} : * \qquad \Gamma \vdash \mathtt{e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \triangleright \mathtt{q}}{\Gamma \vdash \mathtt{fold\ F\ T\ e} : \mu\ \mathtt{F\ T} \triangleright \mathtt{fld\ V\ F\ T\ q}}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T} \triangleright \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \triangleright \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{T} \triangleright \mathtt{app\ V\ T_2\ T\ q_1\ q_2}}$$

$$\frac{\Gamma \vdash \mathtt{F} : (* \to *) \to * \to * \qquad \Gamma \vdash \mathtt{T} : * \qquad \Gamma \vdash \mathtt{e} : \mu\ \mathtt{F\ T} \triangleright \mathtt{q}}{\Gamma \vdash \mathtt{unfold\ F\ T\ e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \triangleright \mathtt{unfld\ V\ F\ T\ q}}$$

$$\frac{\Gamma, (\mathtt{X{:}K}) \vdash \mathtt{e} : \mathtt{T} \triangleright \mathtt{q} \qquad \mathtt{isAll}_{\mathtt{X,K,T}} = \mathtt{p} \qquad \mathtt{stripAll}_{\mathtt{K}} = \mathtt{s} \qquad \mathtt{underAll}_{\mathtt{X,K,T}} = \mathtt{u}}{\Gamma \vdash (\Lambda \mathtt{X{:}K.e}) : (\forall \mathtt{X{:}K.T}) \triangleright \mathtt{tabs\ V\ (\forall X{:}K.T)\ p\ s\ u}\ (\Lambda \mathtt{X{:}K.q})}$$

$$\frac{\langle\rangle \vdash \mathtt{e} : \mathtt{T} \triangleright \mathtt{q}}{\overline{\mathtt{e}} = \Lambda \mathtt{V}{:}* \to *.\ \mathtt{q}}$$

Quotation and pre-quotation

Figure 3.12: Self-representation of $F_{\omega}^{\mu i}$.

```
decl rec evalV : (∀V:∗→∗.∀A:∗. PExp V A → PExp V A) =
 ΛV:∗ → ∗. ΛA:∗. λe:PExp V A.
 matchExp V A e (PExp V A)
  (constVar V A (PExp V A) e)
  (constAbs V A (PExp V A) e)
  (ΛB:∗. λf : PExp V (B → A). λx : PExp V B.
   let f1 : PExp V (B → A) = evalV V (B → A) f in
   let def : PExp V A = app V B A f1 x in
   matchAbs V (B → A) (PExp V A) f1 def
    (ΛB1:∗. ΛA1:∗. λeq : Eq (B1 → A1) (B → A).
     λf : PExp V B1 → PExp V A1.
     let eqL : Eq B B1 = sym B1 B (arrL B1 A1 B A eq) in
     let eqR : Eq A1 A = arrR B1 A1 B A eq in
     let f' : PExp V B → PExp V A =
       eqR (PExp V) ∘ f ∘ eqL (PExp V)
     in evalV V A (f' x)))
  (constTAbs V A (PExp V A) e)
  (ΛB : ∗. λp : IsAll B. λi : Inst B A. λe1 : PExp V B.
   let e2 : PExp V B = evalV V B e1 in
   let def : PExp V A = tapp V B A p i e2 in
   matchTAbs V B (PExp V A) e2 def
    (λp : IsAll B. λs : StripAll B. λu : UnderAll B.
     λe3 : All Id (PExp V) B. evalV V A (i (PExp V) e3)))
  (constFold V A (PExp V A) e)
  (ΛF : (∗ → ∗) → ∗ → ∗. ΛB : ∗.
   λeq : Eq (F (μ F) B) A. λe1 : PExp V (μ F B).
   let e2 : PExp V (μ F B) = evalV V (μ F B) e1 in
   let def:PExp V A = eq (PExp V) (unfld V F B e2) in
   matchFold V (μ F B) (PExp V A) e2 def
    (ΛF1 : (∗ → ∗) → ∗ → ∗. ΛB1 : ∗.
     λeq1 : Eq (μ F1 B1) (μ F B).
     λe3 : PExp V (F1 (μ F1) B1).
     let eq2 : Eq (F1 (μ F1) B1) A =
       trans (F1 (μ F1) B1) (F (μ F) B) A
         (eqUnfold F1 B1 F B eq1) eq
     in evalV V A (eq2 (PExp V) e3)));

decl eval : (∀A:∗. Exp A → Exp A) =
  ΛA:∗. λe:Exp A. ΛV:∗→∗. evalV V A (e V);
```

Figure 3.13: Weak head normal self-evaluator for $F_\omega^{\mu i}$

function `matchAbs`. This is a specialized version of `matchExp` that takes a only one case function, for λ-abstractions, and a default value that is returned in all other cases. The types and definitions of the constant case functions and specialized match functions are given in the appendix. We now turn to the interesting new cases, for reducing type applications and unfold/fold expressions.

When the input represents a type application, we get a proof that `A` is an instance of some quantified type `B`, and the head position subexpression `e1` has type `PExp V B`. If `e1` evaluates to a type abstraction, we get `e3` of type `All Id (PExp V) B`, where `B` is some quantified type. We also know that `A` is an instance of `B`, witnessed by the instantiation function `i` of type `Inst B A`. We use `i` to reduce the redex, instantiating `e3` to `PExp V A`. Then, as before, we continue evaluating the result.

If the input term of type `A` is an `unfold`, then the head position subexpression `e1` has the recursive type `μ F B`, and we get a proof that `A` is equal to the unfolding of `μ F B`. If `e1` evaluates to a `fold`, then we are given proofs that it has a recursive type, which we already knew in this case, and a subexpression `e3` of the unfolded type. The `unfold` expression reduces to `e3`, and we use transitivity to construct a proof to cast `e3` to `PExp V A`, and continue evaluation.

Single-step left-most reduction. Left-most reduction is a restriction of the reduction rules shown in Figure 3.7. It never evaluates under a λ abstraction, a Λ abstraction, or a fold in a redex, and only evaluates the argument of an application if the function is a normal form and the application is not a redex.

Our implementation of left-most reduction has the same type as our weak head evaluator, but differs in that it reduces at most one redex, possibly under abstractions. The top-level function `step` operates only on closed terms. It has the same type as `eval`, `(∀T:∗. Exp T → Exp T)`. The main driver is `stepV`, which has the type `(∀V:∗ → ∗. ∀T:∗. PExp (PExp V) T → PExp V T)`. Its input is a representation of type `PExp (PExp V) T`, which can contain free variables of `PExp V` types. In other words, free variables are themselves representations. This is a key to evaluating under abstractions. When going under an abstraction, we use the `var` constructor to tag the variables, so `stepV` can detect them. As `stepV` walks back out of the

```
PNeExp : (∗ → ∗) → ∗ → ∗
PNfExp : (∗ → ∗) → ∗ → ∗

Sem : (∗ → ∗) → ∗ → ∗

decl NfExp : ∗ → ∗ = λT:∗. ∀V:∗ → ∗. PNfExp V T;

sem   : (∀V:∗→∗. ∀T:∗. Exp T → Sem V T)
reify : (∀V:∗→∗. ∀T:∗. Sem V T → PNfExp V T)

decl nbe : (∀T:∗. Exp T → NfExp T) =
  ΛT:∗. λe:Exp T. ΛV:∗ → ∗.
  reify V T (sem V T e);

unNf : (∀T:∗. NfExp T → Exp T)

decl norm : (∀T:∗. Exp T → Exp T) =
  ΛT:∗. λe:Exp T. unNf T (nbe T e);
```

Figure 3.14: Highlights of our NbE implementation.

representation, it removes the `var` tags.

When evaluating an application, there are three possibilities: either the head subexpression is a λ-abstraction, in which case `stepV` reduces the β-redex, or the head subexpression can take a step, or the head expression is normal, in which case `stepV` steps the argument. `stepV` relies on a normal-form checker to decide whether to step the head or the argument subexpression.

Normalization by evaluation. We can use `step` and a normal-form checker to define a normalizer, by repeatedly stepping a representation until a normal form is reached. This is quite inefficient, though, so we also implement an efficient normalizer using the technique of Normalization by Evaluation (NbE). The implementation of NbE is outlined in Figure 3.14. The top-level function `norm` has the same type `(∀T:∗. Exp T → Exp T)` as `eval` and `step`. The main driver is `nbe`, which maps closed terms to closed normal forms of type `NfExp T`. The type `NfExp` is a PHOAS representation similar to `Exp`, except that it only represents normal form terms. The type `PNfExp` is defined by mutual recursion with `PNeExp`, which represents normal and neutral terms – normal form terms that can be used in head position without introducing a redex. For example, if `f : A` $\to$ `B` is normal and neutral, and `x` is normal, then `f x` is normal and neutral. See Figure 3.7 for a grammar of normal and neutral terms.

We also define a semantic domain `Sem V T` and a function `sem` that `nbe` uses to map representations into the semantic domain. The semantic domain has the property that, if `e1` $\equiv$ `e2`, and `q1` and `q2` are pre-representations of `e1` and `e2` respectively, then `sem V T q1` $\equiv$ `sem V T q2`. The function `reify` maps semantic terms of type `Sem V T` to normal form representations of type `PNfExp V T`. Since normal forms are a subset of expressions, the function `unNf` can convert normal form representations of type `NfExp T` to representations of type `Exp T`.

The type of `nbe` ensures that it maps normalizing terms to their normal form and preserves types. Our `nbe` is not type directed, so it does not produce -long normal forms. This is sometimes called "untyped normalization by evaluation" [40, 6], though this conflicts with our nomenclature of calling a meta-program typed or untyped to indicate whether it operates on typed or untyped abstract syntax. We call our NbE typed, but not type-directed.

3.7 Benchmarks and Experiments

In this section we discuss our benchmark meta-programs, our implementation, and our experiments.

To evaluate the expressive power of our language and representation, we reimplemented the meta-programs from our previous work [20] in $F_\omega^{\mu i}$. We type check and test our evaluators and benchmark meta-programs using an implementation of $F_\omega^{\mu i}$ in Haskell. The implementation includes a parser, type checker, evaluator, and equivalence checker. In particular, we tested that our self-evaluators are self-applicable – they can be applied to themselves.

Benchmark meta-programs. In previous work [20], we implemented a suite of self-applicable meta-programs for F_ω, including a self-interpreter and a continuation-passing-style transformation. We reimplemented all of their meta-programs for $F_\omega^{\mu i}$. They are defined as folds over the representation, so in order to align our reimplementation as closely as possible

```
foldExp :
 ∀R : ∗ → ∗.
 (∀A:∗. ∀B:∗. (R A → R B) → R (A → B)) →
 (∀A:∗. ∀B:∗. R (A → B) → R A → R B) →
 (∀A:∗. IsAll A → StripAll A → UnderAll A →
  All Id R A → R A) →
 (∀A:∗. ∀B:∗. IsAll A → Inst A B → R A → R B) →
 (∀F:(∗→∗)→∗→∗.∀A:∗. R (F (μ F) A) → R (μ F A)) →
 (∀F:(∗→∗)→∗→∗.∀A:∗. R (μ F A) → R (F (μ F) A)) →
 ∀A:∗. Exp A → R A
```

Figure 3.15: Interface for defining folds over representations.

to the originals, we also implemented a general fold function for our representation.

Figure 3.15 shows the type of our general purpose `foldExp` function. It is a recursive function that takes six fold functions, one for each form of expression other than variables, which are applied uniformly throughout the representation. The type `R` determines the result type of the fold. We also instantiate `V` to `R`, so we can use `var` to embed partial results of the fold into the representation.

The type of `foldExp` is reminiscent of the `Exp` type used in our previous work [20], which is defined by its fold. Notable differences are the addition of fold functions for `fold` and `unfold`, and our improved treatment of polymorphic types using `Typecase`.

We implemented a self-recognizer `unquote` that recovers a term from its representation. It has the type `∀A:∗. Exp A → A`, and is defined by a fold with `R=Id`, the identity function on types. `unquote` uses `IsAll` proofs in a way we haven't seen so far. The fold function for type abstractions gets a term of type `All Id Id A`. When `A` is a concrete quantified type `∀X:K.T`, this is equivalent to `A`. However, the fold function is defined for an abstract quantified type `A`. It uses the `UnAll A` component of an `IsAll A` proof to convert `All Id Id A` to `A`.

A continuation-passing-style (CPS) transformation makes evaluation order explicit and gives a name to each intermediate value in the computation. It also transforms the type of the term in a nontrivial way – the result type is expressed as a recursive function on the input type. The type of our typed call-by-name CPS transformation is shown in Figure 3.16. Previous implementations of typed CPS transformation [71, 19, 20] use type-level representations of types in order to express this relationship. The type representations were designed to support the kind of function needed to typecheck CPS. A challenge of this approach is that the encoded types should have the same equivalences as regular types. That is, if two types `A` and `B` are equivalent, then their encodings should be as well. In previous work, we used a nonstandard encoding of types to ensure this [20].

In this work, we do not encode types. Instead we combine recursive types and `Typecase` in a new way to express the type of our CPS transformation. Intuitively, `CPS` is an iso-recursive intensional type function. The specification for the type `CPS` is given below, and its definition in $F_\omega^{\mu i}$ is shown in Figure 3.16. `T1` ≅ `T2` denotes that the types `T1` and `T2` are isomorphic, witnessed by `unfold` and `fold`. A value of type `Ct T` is function that takes a continuation and calls that continuation with an argument of type `T`.

```
CPS (A → B)  ≅ Ct (CPS A → CPS B)
CPS (∀X:K.T) ≅ Ct (∀X:K. CPS T)
CPS (μ F A)  ≅ Ct (CPS (F (μ F) A))
```

Like `unquote`, `cps` uses `IsAll` proofs in an interesting new way. It is defined as a fold, and the case function for type abstractions is given an `All Id CPS A`, which it needs to cast to `CPS1F CPS1 A`, the unfolding of `CPS1 A`. `All Id CPS A` and `CPS1F CPS1 A` are both `Typecase` types, and while their cases for quantified types are the same, the cases for arrow types and recursive types are different. This is where the `TcAll A` component of the `IsAll A` proof is useful. Since we know `A` is a quantified types, the `Typecase` cases for arrow types and recursive types are irrelevant. The function `eqCPSAll` uses `TcAll A` to prove `CPS1F CPS1 A` and `All Id CPS A` are equal.

```
decl eqCPSAll : (∀A:∗. IsAll A →
                 Eq (CPS1F CPS1 A) (All Id CPS A)) =
  ΛA:∗. λp : IsAll A.
  fst p (λA1:∗.λA2:∗. CPS A1 → CPS A2)
        Id CPS
        (λF:(∗→∗)→∗→∗.λB:∗. CPS (F (μ F) B));
```

We also implement the other meta-programs from our previous work: a size measure, a normal form checker, and a top-level syntactic form checker. The complete code for all our

```
decl Ct : ∗ → ∗ = λA:∗. ∀B:∗. (A → B) → B;
decl CPS1 : ∗ → ∗ =
 µ (λCPS1:∗ → ∗. λT:∗.
    Typecase
     (λA:∗. λB:∗. Ct (CPS1 A) → Ct (CPS1 B))
     Id (λT:∗. Ct (CPS1 T))
     (λF:(∗→∗)→∗→∗. λT:∗. Ct (CPS1 (F (µ F) T)))
     T);
decl CPS : ∗ → ∗ = λT:∗. Ct (CPS1 T);

cps : (∀T:∗. Exp T → CPS T)
```

Figure 3.16: Type of our CPS transformation

meta-programs is provided in the appendix. The `size` measure demonstrates the use of our `strip` functions to remove redundant quantifiers. Below is the fold function given to `foldExp` for type abstractions:

```
decl sizeTAbs : FoldTAbs (λT:∗. Nat) =
 ΛA:∗. λp:IsAll A. λs:StripAll A.
 λu:UnderAll A. λf:All Id (λT:∗.Nat) A.
 succ (s Nat f);
```

Here, `A` is some unknown quantified type, and `f` holds the result of the recursive call to `size` on the body of the type abstraction. The size of the type abstraction is one more than the size of its body, so `sizeTAbs` needs to apply the successor function to the result of the recursive call. However, its type `All Id (λT:∗.Nat) A` is different than `Nat`. For example, if `A = (∀X:K.A′)`, then `All Id (λT:∗.Nat) A ≡ (∀X:K.Nat)`. The quantifier on `X` is redundant, and blocks `sizeTAbs` from accessing the result of the recursive call. By removing the redundant quantifier, the strip function `s` is instrumental in programming `size` on representations of polymorphic terms.

Implementation. We have implemented System $F_\omega^{\mu i}$ in Haskell. The implementation includes a parser, type checker, quoter, evaluator (which does the *evaluation* in Figure 3.1), and an equivalence checker. Our evaluator is based on NbE similar to Figure 3.14, except that it operates on untyped first-order abstract syntax based on DeBruijn indices. Our self-evaluators and other meta-programs have been implemented, type checked and tested. Our parser includes special syntax for building quotations and normalizing terms, which is useful for testing. We use `[e]` to denote the representation of `e`, and `<e>` to denote the normal form of `e`. The normalization of `<e>` expressions occurs after type checking, but before quotation. Thus `[<e>]` denotes the representation of the normal form of `e`.

We test our meta-programs using functions on natural numbers, which use all the features of the language: recursive types, recursive functions, and polymorphism. We encode natural numbers using a typed Scott encoding [3, 96] that is similar to our encoding of $F_{\omega}^{\mu i}$ terms. Compared to other encodings, Scott-encoded natural numbers support natural implementations of functions like predecessor, equality, and factorial.

We use our equivalence checker to test our meta-programs. It works by normalizing the two terms, and checking the results for syntactic equality up to renaming. For example, we test that our implementation of NbE normalizes the representation `[fact five]` to the representation of its normal form, `[<fact five>]`:

```
norm Nat [fact five] ≡ [<fact five>]
```

Self-application. Each of our evaluators is *self-applicable*, meaning that it can be applied to its own representation. In particular, the self-application of `eval` is written `eval (∀T:∗. Exp T → Exp T) [eval]`. We have self-applied each of our evaluators, and tested the result. Here is an example, specifically for our weak head normal form evaluator:

```
decl eval' = unquote (∀T:∗. Exp T → Exp T)
              (eval (∀T:∗. Exp T → Exp T) [eval]);

eval' Nat [fact five] ≡ eval Nat [fact five]
```

We define `eval'` by applying `eval` to its representation `[eval]`, and then unquoting the result. In terms of Figure 3.1, we start with `eval` at the bottom-left corner, then move up to its representation `[eval]`, then right to the representation of its value (weak head normal form in this case) `(eval (∀T:∗. Exp T → Exp T) [eval])`, and `unquote` recovers the value from its representation. Finally test that `eval'` and `eval` have the same behavior by testing that they map equal inputs to equal outputs.

3.8 Related Work

Typed self-representation. Pfenning and Lee [69] considered whether System F could support a useful notion of a self-interpreter, and concluded that the answer seemed to be "no". They presented a series of typed representations, of System F in F_ω, and of F_ω in F_ω^+, which extends F_ω with kind polymorphism. Whether typed self-representation is possible remained an open question until 2009, when Rendel, Ostermann and Hofer [71] presented the first typed-self representation. Their language was a typed λ-calculus F_ω^* that has undecidable type checking. They implemented a self-recognizer, but not a self-evaluator. Jay and Palsberg [52] presented a typed self-representation for a combinator calculus that also has undecidable type checking. Their representation supports a self-recognizer and a self-evaluator, but not with the types described in Section 1. In their representation scheme, terms have the same type as their representations, and both their interpreters have the type $\forall \mathtt{T}.\ \mathtt{T} \to \mathtt{T}$. In previous work we presented self-representations for System U [19], the first for a language with decidable type checking, and for F_ω [20], the first for a strongly normalizing language. Each of these supported self-recognizers and CPS transformations, but not self-evaluators.

There is some evidence that the problem of implementing a typed self-evaluator is more difficult than that of implementing a typed self-recognizer. For example, self-recognizers have been implemented in simpler languages than $F_\omega^{\mu i}$, and based on simpler representation techniques. A self-recognizer implemented as a fold relies entirely on meta-level evaluation. The fact that meta-level evaluation is guaranteed to be type-preserving simplifies the implementation of a typed self-recognizer, but the evaluation strategy can only be what the meta-level implements. On the other hand, self-evaluators can control the evaluation strategy, but this requires more work to convince the type checker that the evaluation is type-preserving (e.g. by deriving type equality proofs).

Typed self-evaluation is an important step in the area of typed self-representation. It lays the foundation for other verifiably type-preserving program transformations, like partial evaluators or program optimizers. Our representation techniques can be used to explore for other applications such as typed Domain Specific Languages (DSLs), typed reflection, or multi-stage programming.

It remains an open problem to implement a self-evaluator for a strongly normalizing language without recursion. We use recursion in two ways in our evaluators: first, we use a recursive type for our representation, which has a negative occurrence in its `abs` constructor. Second, we use the fixpoint combinator to control the order of evaluation. This allows our evaluators to select a particular redex in a term to reduce. Previous work on typed-self representation only supported folds, which treat all parts of a representation uniformly.

Intensional type analysis. Intensional type analysis (ITA) was pioneered by Harper and Morrisett [50] for efficient implementation of ad-hoc polymorphism. Previous work on intensional type analysis has included an ITA operator in terms as well as types. Term-level ITA enables runtime type introspection (RTTI), and the primary role of type-level ITA has traditionally been to typecheck RTTI. RTTI is useful for dynamic typing [93], typed compilation[64, 33], garbage collection [86], and marshalling data [38]. ITA has been shown to support type-erasure semantics [33, 34], user-defined types [89], and a kind of parametricity theorem [67].

Early work on ITA was restricted to monotypes – base types, arrows, and products[50]. Subsequently, it was extended to handle polymorphic types [33], higher-order types [94], and recursive types [31]. Trifonov et al. presented λ_i^Q [86], which supports *fully-reflexive* ITA – analysis of all types of kind $*$, including quantified and recursive types.

The most notable difference between $F_\omega^{\mu i}$ and previous languages with ITA is that $F_\omega^{\mu i}$ does not include a term-level ITA operator, and thus does not support runtime type introspection. Our type-level `Typecase` operator is fully-reflexive, but we restrict the analysis on quantified types to avoid kind-polymorphism, which was used in λ_i^Q. Unlike our `Typecase` operator, the type-level ITA operator in λ_i^Q is recursive, which requires more complex machinery to keep type checking decidable.

Our `Typecase` operator is simpler than those from previous work on intensional type analysis. Also, by omitting the term-level ITA operator, we retain a simple semantics of $F_\omega^{\mu i}$.

In particular, the reduction of terms does not depend on types. This in turn simplifies our presentation, our self-evaluators and the proofs of our meta-theorems.

GADTs. Generalized algebraic data types (GADTs) were introduced independently by Cheney and Hinze [28] and Xi, Chen and Chen [98]. They applications include intensional type analysis [28, 98, 90] and typed meta-programming [48]. Traditional formulations of GADTs are designed to support efficient encodings, pattern matching, type inference, and/or type erasure semantics [98, 82]. In this work our focus has been to identify a core calculus and representation techniques that can support typed self-evaluation. While our representation is conceptually similar to a GADT, it is meta-theoretically much simpler than a traditional GADT. More work is needed to achieve self-representation and self-evaluation for a full language that includes efficient implementations of GADTs. One important question that needs to be answered is how to represent and evaluate programs that involve user-defined GADTs. For example, if we used a GADT for our self-representation, how would we represent the self-evaluators that operate on it?

Type equality. Type equality has been used to encode GADTs [60, 28, 98, 82], and for generic programming [99, 27], dynamic typing [9, 27, 93], typed meta-programming [79, 68], and simulating dependent types [26]. Some formulations of type equality are built-into the language in order to support type-erasure semantics [82] and type inference [98, 82, 79]. This comes at a cost of a larger and more complex language, which makes self-interpretation more difficult.

The use of polymorphism to encode Leibniz equality [9, 93, 27] is perhaps the simplest encoding technique, though it lacks support for erasure (leading to some runtime overhead) and type inference. Furthermore, without intensional type functions Leibniz equality is not expressive enough for defining typed evaluators, a limitation we have addressed in this chapter. Our formulation of type equality has essentially no impact on the semantics, because the heavy lifting is done at the type level by `Typecase`.

3.9 Conclusion

We have presented $\mathrm{F}_{\omega}^{\mu i}$, a typed λ-calculus with decidable type checking, and the first language known to support typed self-evaluation. We use intensional type functions to implement type equality proofs, which we then use to define a typed self-representation in the style of Generalized Algebraic Data Types (GADTs). Our three polymorphically-typed self-evaluators implement weak head normal form evaluation, single-step left-most β-reduction, and normalization by evaluation (NbE). Our self-representation also supports all the benchmark meta-programs from previous work on typed self-representation.

We leave for future work the question of whether typed self-evaluation is possible for a language with support for efficient user-defined types.

CHAPTER 4

Typed Self-Applicable Partial Evaluation

4.1 Introduction

A partial evaluator is a meta-program that specializes another program to some of its inputs. When given the remaining inputs, the specialized program will compute the same result as running the original program on all its inputs. Kleene's s-m-n theorem established that such a specialization process is possible, and since then partial evaluation has been established as a practical technique for program optimization and automatic program generation.

We will define partial evaluation as the specialization of a two-input function to its first input (corresponding to the s-1-1 instance of s-m-n).

Definition 4.1.1 (Partial Evaluation). `mix` *is a partial evaluator if for every program* `p` *and input* `x`*, there exists a specialized program* $\mathtt{p_x}$ *such that:*

1. *If* $\mathtt{mix}\ \overline{\mathtt{p}}\ \overline{\mathtt{x}}$ *has a normal form, that normal form is* $\overline{\mathtt{p_x}}$
2. $\forall \mathtt{y}.\ \mathtt{p_x}\ \mathtt{y} \equiv_\beta \mathtt{p}\ \mathtt{x}\ \mathtt{y}$

The first condition states that the partial evaluator may not be total: specialization may not terminate. When it does terminate, however, it outputs a specialized program $\mathtt{p_x}$. The second condition states that the specialized program $\mathtt{p_x}$ is correct: it has the same behavior as `p x` on all inputs `y`. The definition naturally extends to other numbers of inputs. The inputs to which the program is specialized are often called "static" and the remaining inputs "dynamic".

A partial evaluators is *self-applicable* if it can specialize itself. Futamura showed that self-applicable partial evaluation relates compilation and interpretation, and can automati-

Description	Definition	Correctness Specification
1. Compile S-program to T	$\texttt{mix}\ \overline{\texttt{int}}\ \overline{\texttt{s}} \equiv_\beta \overline{\texttt{t}}$	$\forall\texttt{x}.\ \texttt{t x} \equiv_\beta \texttt{int}\ \overline{\texttt{s}}\ \texttt{x}$
2. Generate an S-to-T compiler	$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\texttt{int}} \equiv_\beta \overline{\texttt{comp}}$	$\forall\texttt{s}.\ \texttt{comp}\ \overline{\texttt{s}} \equiv_\beta \texttt{mix}\ \overline{\texttt{int}}\ \overline{\texttt{s}}$
3. Generate a compiler-generator	$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\texttt{mix}} \equiv_\beta \overline{\texttt{cogen}}$	$\forall\texttt{int}.\ \texttt{cogen}\ \overline{\texttt{int}} \equiv_\beta \texttt{mix}\ \overline{\texttt{mix}}\ \overline{\texttt{int}}$
4. Self-generation of `cogen`		$\texttt{cogen}\ \overline{\texttt{mix}} \equiv_\beta \overline{\texttt{cogen}}$

Figure 4.1: The four Futamura projections.

cally generate compilers and compiler-generators from interpreters[41]. The four Futamura projections are shown in Figure 4.1. For each projection we list a correctness specification implied by the correctness of `mix`.

The first Futamura projection compiles programs from a source language S to a target language T by specializing an S-interpreter programmed in T. Given any input to the program, the target program gives the same result as interpreting the source program. The second projection generates a compiler from S-programs to T-programs by specializing the partial evaluator `mix` to the interpreter. Given any program, the compiler gives the same result as the first projection. The third projection generates a compiler-generator by specializing `mix` to itself. Given any interpreter, the compiler-generator gives the same result as the second projection. Futamura only discussed these three Futamura projections. Subsequently Glück [46] showed that there are more. In particular, the fourth projection demonstates that the compiler-generator `cogen` can generate itself when applied to `mix`. This is sometimes called the "mixpoint".

Until now we've left unsaid the purpose for specialization: to make programs run more efficiently. If $\texttt{p}_\texttt{x}$ runs no faster than `p x`, we may as well skip specialization and use `p x` itself. In fact, a trivial partial evaluator could simply define $\texttt{p}_\texttt{x}$ to be `p x`. How can we determine that a partial evaluator is non-trivial – that it provides some benefit to efficiency? Jones, Gomard and Sestoft [54] offer a criterion, widely known as Jones-optimality, that is based on the first Futamura projection. Intuitively, the idea is that specialization should remove all the "interpretative overhead" of a self-interpreter. Compilation by using first Futamura projection with a self-interpreter allows us to directly compare the performance of the source and target programs, since the source and target languages are the same. Jones-optimality states that no target program should be less efficient than its source program – on any input.

This has become the gold standard for non-trivial partial evaluation.

In this chapter, we present the first typed self-applicable partial evaluator. It is Jones-optimal and can be used to generate the four Futamura projections. Our starting point is Mogensen's self-applicable partial evaluator for the untyped λ-calculus [62]. Mogensen showed that a self-evaluator that evaluates to β-normal form can be used as the basis for a self-applicable partial evaluator – a technique we call "specialization by normalization". We use our typed self-evaluator `nbe` from Chapter 4 to implement a typed version of Mogensen's partial evaluator in $F_\omega^{\mu i}$. The result is self-applicable and generates the Futamura projections, but is not Jones-optimal. Worse, specialization by normalization sometimes generates specialized code that is *slower* than the original. To solve this, we present some simple modifications that guarantee our partial evaluator never causes a slowdown and make it Jones-optimal.

In section 2 we discuss the possible types of self-applicable partial evaluators and the Futamura projections, in section 3 we present our typed version of Mogensen's partial evaluator, in section 4 we discuss why Mogensen's partial evaluator is not Jones-optimal and how to achieve optimality, in section 5 we present the changes we make to the self-representation of $F_\omega^{\mu i}$, in section 6 we present the details of our partial evaluator, in section 7 we discuss our experimental results, in section 8 we compare with related work, and in section 9 we conclude.

4.2 The Type of a Self-Applicable Partial Evaluator

What is the type of a self-applicable partial evaluator? Jones, Gomard and Sestoft [54, Section 16.2] proposed the type:

```
∀A:*. ∀B:*. Exp (A → B) → A → Exp B        (†)
```

They showed that a partial evaluator with this type can be self-applied and used to derive the Futamura projections. However, the realization of a typed self-applicable partial evaluator with this type has remained an open problem. There is a limitation to this type: the second

Description	Definition	Correctness Specification
1. Compile S-program to T	$\texttt{mix}\ \overline{\texttt{int}}\ \overline{\overline{\texttt{s}}} \equiv_\beta \overline{\texttt{t}}$	$\forall\texttt{x}.\ \texttt{t}\ \texttt{x} \equiv_\beta \texttt{int}\ \overline{\texttt{s}}\ \texttt{x}$
2. Generate an S-to-T compiler	$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{int}}} \equiv_\beta \overline{\texttt{comp}}$	$\forall\texttt{s}.\ \texttt{comp}\ \overline{\texttt{s}} \equiv_\beta \texttt{mix}\ \overline{\texttt{int}}\ \overline{\overline{\texttt{s}}}$
3. Generate a compiler-generator	$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{mix}}} \equiv_\beta \overline{\texttt{cogen}}$	$\forall\texttt{int}.\ \texttt{cogen}\ \overline{\texttt{int}} \equiv_\beta \texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{int}}}$
4. Self-generation of `cogen`		$\texttt{cogen}\ \overline{\texttt{mix}} \equiv_\beta \overline{\texttt{cogen}}$

Figure 4.2: The Futamura projections for a Mogensen-style `mix`.

input to the partial evaluator has a completely abstracted type. In a typed language without reflection, the partial evaluator can make no assumptions about the input, and cannot inspect or manipulate it.

The type (†) is also inconsistent with the types of representations in our system. To demonstrate, let's consider an example application of Mogensen's partial evaluator. The application $\texttt{mix}\ \overline{\texttt{ackermann}}\ \overline{\texttt{zero}}$ specializes the Ackermann function to the Church numeral 0. If `ackermann` has the type $\texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$ and `zero` has the type `Nat`, then under a typed representation scheme the representation $\overline{\texttt{ackermann}}$ would have the type $\texttt{Exp (Nat} \to \texttt{Nat} \to \texttt{Nat)}$ and the representation $\overline{\texttt{zero}}$ would have the type `Exp Nat`. There is no instantiation of the type (†) that can accept inputs of those types.

We adopt the following type, which has been used before as the type of a partial evaluator [22] and is consistent with the types of our representations:

$$\forall\texttt{A:*}.\ \forall\texttt{B:*}.\ \texttt{Exp (A} \to \texttt{B)} \to \texttt{Exp A} \to \texttt{Exp B} \qquad (\ddagger)$$

This type allows the partial evaluator to rely on its second argument being a representation. This is important because the second argument (or a part of it) could end up in the residual code returned by the partial evaluator, in which case it must be converted to a representation. This would not be possible in general if the second argument could be any term.

A consequence of this type is that the Futamura projections involve double-representation: representations of representations. This is shown in Figure 4.2. For example, the first Futamura projection specializes an interpreter of type $\texttt{(Exp T} \to \texttt{T)}$ to an input of type `Exp T`. If we instantiate the type (‡) with `A := Exp T` and `B := T`, we get $\texttt{(Exp (Exp T)} \to \texttt{T)} \to \texttt{Exp (Exp T)} \to \texttt{Exp T}$. The type `Exp (Exp T)` is the type of a double-representation. Double-representation can lead to nontrivial overheads that threaten to dominate the benefits of self-application. However, a sufficiently powerful partial evaluator can potentially eliminate that overhead. For example, the generated compiler `comp` produces the same output as the first Futamura projection, but operates on ordinary representations instead of double-representations.

Figure 4.3 shows typed versions of the Futamura projections in the concrete syntax of $F_\omega^{\mu i}$. Quotations are denoted using square brackets `[-]` rather than overlines. The first projection compiles the factorial function `fact` by specializing the self-interpreter `unquote` to it. The second projection generates a compiler by specializing `mix` to `unquote`. The third projection generates a compiler-generator by specializing `mix` to itself. The fourth projection generates the self-generating compiler-generator. It is also obtained by specializing `mix` to itself, but at different types.

4.3 Typed Specialization by Normalization

Mogensen implemented the first self-applicable partial evaluator for (untyped) λ-calculus. He recognized that a self-evaluator that evaluates to β-normal form can used as the basis for a partial evaluator. We call this approach "specialization by normalization". With this approach, specializing a program `f` to an input `x` with such a partial evaluator with result in the normal form of `f x`. One problem with specialization by normalization is that not all terms have a normal form, which leads to non-termination of the partial evaluator: if `f x` has no normal form, then specializing `f` to `x` will not terminate. In particular, if the self-interpreter `unquote` and the partial evaluator `mix` do not have normal forms, then the Futamura projections will not terminate. Mogensen solves this problem by using a representation based on Church-encoding, for which `unquote` and `mix` do have normal forms. The partial evaluator is self-applicable and the Futamura projections terminate.

To summarize, Mogensen made three key insights: 1) a self-evaluator that evaluates to normal form can be used as the basis for a partial evaluator (specialization by normalization), 2) specialization by normalization supports the Futamura projections if the self-interpreter and partial evaluator themselves are strongly normalizing, and 3) a representation based

```
decl fact_compiled : Exp (Nat → Nat) =
  mix (Exp (Nat → Nat)) (Nat → Nat)
    [unquote (Nat → Nat)]
    [[fact]]

decl compile : ∀A:*. Exp (Exp (Exp A) → Exp A) =
  ΛA:*.
  mix (Exp (Exp A → A)) (Exp (Exp A) → Exp A)
    [mix (Exp A) A]
    [[unquote A]]

decl cogen : (∀A:*. ∀B:*. Exp (Exp (Exp (A → B)) →
                               Exp (Exp A → Exp B))) =
  ΛA:*. ΛB:*.
  mix (Exp (Exp (A → B) → Exp A → Exp B))
      (Exp (Exp (A → B)) → Exp (Exp A → Exp B))
      [mix (Exp (A → B)) (Exp A → Exp B)]
      [[mix A B]]

decl selfgen : (∀A:*. ∀B:*.
                Exp (Exp (Exp (A → B) → Exp A → Exp B)) →
                     Exp (Exp (Exp (A → B)) →
                          Exp (Exp A → Exp B)))) =
  ΛA:*. ΛB:*.
  mix (Exp (Exp (Exp (A → B) → Exp A → Exp B) →
       Exp (Exp (A → B)) →
       Exp (Exp A → Exp B)))
      (Exp (Exp (Exp (A → B) → Exp A → Exp B)) →
       Exp (Exp (Exp (A → B)) → Exp (Exp A → Exp B)))
   [mix (Exp (Exp (A → B) → Exp A → Exp B))
        (Exp (Exp (A → B)) → Exp (Exp A → Exp B))]
   [[mix (Exp (A → B)) (Exp A → Exp B)]]
```

Figure 4.3: The four typed Futamura projections

```
decl mix : (∀A:*. ∀B:*. Exp (A → B) → Exp A → Exp B) =
  ΛA:*. ΛB:*. λe1 : Exp (A → B). λe2 : Exp A.
  nbe B (ΛV:* → *. app V A B (e1 V) (e2 V))
```

Figure 4.4: A typed specialization-by-normalization partial evaluator

on Church-encoding supports a strongly normalizing self-interpreter and specialization by normalization partial evaluator.

In this section, we present a typed specialization by normalization partial evaluator for $F_\omega^{\mu i}$. To support the Futamura projections, we follow Mogensen's approach and move to a typed self-representation of $F_\omega^{\mu i}$ based on Church-encoding (also called a tagless-final or Böhm-Berarducci encoding) similar to the representation of F_ω described in Chapter 2. After porting our self-interpreter and normalizing self-evaluator to this representation, they are both strongly normalizing.

Our typed, self-applicable partial evaluator for $F_\omega^{\mu i}$ based on Mogensen's specialization by normalization approach is shown in Figure 4.4. Given representations of a program `f` and an input `x`, the computes the specialization of `f` to `x` by normalizing the representation of the application `f x`. This version of `mix` has the type (‡) discussed in Section 4.2. The four Futamura projections have the types and definitions shown in Figure 4.1, and they all normalize and meet their specifications.

We compared the speedups obtained from our typed version of Mogensen's partial evaluator with the original untyped version, by counting the number of reductions to normalize unspecialized and specialized versions of a number of Mogensen's benchmark programs. To facilitate a direct comparison, we first erase $F_\omega^{\mu i}$ terms to untyped λ-calculus. This corresponds to a type-erasure semantics, as is used in many practical statically-typed languages.

Program	**Specializer**	**Baseline Steps**	**Specialized Steps**	**Speedup**
`Ackermann 0 3`	untyped	7	3	2.33
	typed	7	3	2.33
`Ackermann 1 3`	untyped	19	13	1.46
	typed	19	13	1.46
`Ackermann 2 3`	untyped	76	65	1.17
	typed	76	65	1.17
`Ackermann 3 3`	untyped	3685	3621	1.02
	typed	3685	3621	1.02
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{0 3}$	untyped	21	7	3
	typed	58	7	8.29
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{1 3}$	untyped	53	19	2.79
	typed	175	19	9.21
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{2 3}$	untyped	208	76	2.74
	typed	749	76	9.86
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{3 3}$	untyped	9866	3685	2.68
	typed	36027	3685	9.78
$\texttt{mix}\ \overline{\texttt{unquote}}\ \overline{\overline{\texttt{Ackermann}}}$	untyped	382	337	1.13
	typed	3439	1562	2.20
$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{unquote}}}$	untyped	1539	334	4.61
	untyped	377029	113546	3.32
$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{mix}}}$	untyped	268481	115931	2.32
	typed	18762389	5423355	3.46

Table 4.1: Speedup from typed vs untyped specialization by normalization

Definition 4.3.1 (Type erasure). *The type erasure of a term* $\mathtt{e}$ *is defined recursively as follows:*

1. $erase(\mathtt{x}) = \mathtt{x}$
2. $erase(\lambda\mathtt{x}{:}\mathtt{T}.\ \mathtt{e}) = \lambda\mathtt{x}.\ erase(\mathtt{e})$
3. $erase(\mathtt{e}_1\ \mathtt{e}_2) = erase(\mathtt{e}_1)\ erase(\mathtt{e}_2)$
4. $erase(\Lambda\mathtt{X}{:}\mathtt{K}.\mathtt{e}) = erase(\mathtt{e})$
5. $erase(\mathtt{e}\ \mathtt{T}) = erase(\mathtt{e})$
6. $erase(\mathtt{fold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}) = erase(\mathtt{e})$
7. $erase(\mathtt{unfold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}) = erase(\mathtt{e})$

We measure the time to evaluate an erased term by counting the number of β-reduction steps required to reach a β-normal form. This number (and whether a normal form is reached at all) will depend on the evaluation strategy used. In this section our purpose is to compare with Mogensen, so we use lazy or call-by-need evaluation, the same strategy he used. In Section 4.7 we also consider call-by-value and call-by-name evaluation.

The results are shown in Table 4.1. The first four rows show speedups obtained by specializing Ackermann's function to its first input. For these, our speedups match Mogensen's exactly. In fact, our Ackermann function erases exactly to Mogensen's, and each of our specialization of it erases to Mogensen's corresponding specialization. The next four rows show the overhead of interpreting Ackermann's function on the same inputs, and how much of that overhead can be removed by specialization. The interpretational overhead is larger for our typed version, because our language is larger – there is more syntax in $F_\omega^{\mu i}$, and therefore more cases in its self-representation and interpreter than in those for untyped λ-calculus. The extra cases lead to larger representations and extra interpretational work that is not removed by type erasure. However, we can specialize away all the extra interpretational overhead: the Ackermann function has the same performance whether compiled by either the untyped or typed version of `mix`. The next (ninth) line compares the cost of compiling the Ackermann function. The baseline version compiles via the first Futamura projection, and the specialized version uses the compiler generated by the second Futamura projection. The results show that the typed version is slower, but yields greater speedups from the second

Futamura projection. The tenth line compares the cost of generating a compiler for untyped λ-calculus versus $F_\omega^{\mu i}$. The baseline version generates via the second Futamura projection, and the specialized version uses the compiler-generator from the third Futamura projection. This time, the untyped version is faster and has greater speedups. The last line compares the cost of generating a compiler generator for untyped λ-calculus versus $F_\omega^{\mu i}$. The baseline version generates via the third Futamura projeciton, and the specialized version uses the self-generating compiler-generating from the fourth Futamura projection. As usual the untyped version is faster, and this time the typed version yields a greater speedup.

The core of the partial evaluator in Figure 4.4 is the self-evaluator `nbe`. This design is modular: we can implement more partial evaluators by replacing `nbe` with other evaluators. In this view, specialization by normalization is an instance of a more general approach we could call specialization by evaluation. Each partial evaluator defined in this way will be self-applicable if the evaluator is a self-evaluator. This more general approach was used by Carette et al. [22] to implement a typed partial evaluator for a simply-typed language in MetaOCaml. We can implement more self-applicable partial evaluators for $F_\omega^{\mu i}$ by replacing `nbe` with a different self-evaluator. This establishes a spectrum of specializers, with specialization by normalization at one end. At the other end is the trival evaluator that does nothing – in effect, the identity function on representations. This results in a trivial specializer that does nothing other than remember the specialized input: the specialization of `f` to `x` is just the application `f x`.

The trivial partial evaluator is not particularly useful – since its evaluator does no evaluation, it provides no speedup and is not Jones-optimal. One the other hand, specialization by normalization is not Jones optimal either. As we will see in Section 4.5, `nbe` does *too much* evaluation. As a result, specialization sometimes causes a slowdown. The reason is that not all β-redexes should be reduced at specialization time, because they might cause the specialized code to run slower than the original. In Section 4.5 we present a self-evaluator that only reduces those redexes that will not produce slower specialized code. The partial evaluator derived from it lies between the two extremes of the specialization by evaluation spectrum. It is Jones-optimal and never causes a slowdown.

4.4 Representation

In this section, we discuss changes we made to the self-representation of $F_\omega^{\mu i}$, the quoter, and the self-interpreter `unquote` to support self-applicable partial evaluation and the generation of the Futamura projections. The result is a combination of representation techniques from Chapters 2 and Chapter 3: We use a Böhm-Berarducci or tagless-final encoding similar to the one for F_ω in Chapter 2, and we the use type equality proofs from Chapter 3. As usual, the representation is an instance of Parametric Higher-Order Abstract Syntax (PHOAS) [29]. The new representation and quotation procedure are defined in Figure 4.5.

The type `PExp` to typecheck pre-quotations, and `Exp` is used to typecheck quotations. The types `Abs`, `App`, `TAbs`, `TApp`, `Fold` and `Unfold` are the types of the case functions used to construct pre-quotations.

Our pre-quoter $\Gamma \vdash$ `e : T` $\triangleright$ `q` maps a typing judgement $\Gamma \vdash$ `e : T` to a pre-quotation `q`. As usual, a variable is represented by another variable with the same name but with a different type. Each other syntactic form of terms is represented by recursively pre-quoting and applying a case function for that form. To represent type abstractions and applications, we use the familiar utility functions that constitute our extensional representation technique for polymorphic terms: instantiation functions that instantiate a polymorphic term, strip functions that remove a redundant quantifier, and under functions that apply can apply functions under a quantifier. Quotation is defined by first pre-quoting the term, then abstracting over the case functions and the type parameter `V` that determines the type of variables.

The tagless-final representation technique encodes a fold over the term. This eliminates the need to implement folds using general recursion as we did in Chapter 3 with the `foldExp` function. However, not all operations are naturally expressed or easily type-checked as a fold. For these, we can convert back from the tagless-final representation to the Mogensen-Scott representation used in Chapter 3.

The self-interpreter `unquote` for the tagless-final representation is shown in Figure 4.6. It is similar to the self-interpreter for F_ω defined in Chapter 2 and the one for the Mogensen-Scott representation of $F_\omega^{\mu i}$ discussed in Chapter 3. We compare with the F_ω version from

```
decl PExp : (∗ → ∗) → ∗ → ∗ =
  λV : ∗ → ∗. λA:∗.
  (∀S:∗. ∀T:∗. (V S → V T) → V (S → T)) →
  (∀A:∗. ∀B:∗. V (A → B) → V A → V B) →
  (∀A:∗. IsAll A → StripAll A → UnderAll A → All Id V A → V A) →
  (∀A:∗. ∀B:∗. IsAll A → Inst A B → V A → V B) →
  (∀F : (∗ → ∗) → ∗ → ∗. ∀B : ∗. V (F (μ F) B) → V (μ F B)) →
  (∀F : (∗ → ∗) → ∗ → ∗. ∀B : ∗. V (μ F B) → V (F (μ F) B)) →
  V A

decl Exp : ∗ → ∗ = λA:∗. ∀V:∗ → ∗. PExp V A

decl Abs  V = ∀A1:∗. ∀A2:∗. (V A1 → V A2) → V (A1 → A2)
decl App  V = ∀A:∗. ∀B:∗. V (A → B) → V A → V B
decl TAbs V =
  ∀A:∗. IsAll A → StripAll A → UnderAll A → All Id V A → V A
decl TApp V = ∀A:∗. ∀B:∗. IsAll A → Inst A B → V A → V B
decl Fold V =
  ∀F : (∗→∗) → ∗ → ∗. ∀B:∗. V (F (μ F) B) → V (μ F B)
decl Unfold V =
  ∀F : (∗→∗) → ∗ → ∗. ∀B : ∗. V (μ F B) → V (F (μ F) B)
```

Definitions of `PExp` and `Exp`

$$\frac{(\mathtt{x} : \mathtt{T}) \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \triangleright \mathtt{x}}$$

$$\frac{\Gamma \vdash \mathtt{T1} : * \qquad \Gamma, (\mathtt{x}:\mathtt{T1}) \vdash \mathtt{e} : \mathtt{T2} \triangleright \mathtt{q}}{\Gamma \vdash (\lambda \mathtt{x}:\mathtt{T1}.\mathtt{e}) : \mathtt{T1} \to \mathtt{T2} \triangleright \mathtt{abs\ T1\ T2}\ (\lambda \mathtt{x}:\mathtt{V\ T1.\ q})}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T} \triangleright \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \triangleright \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{T} \triangleright \mathtt{app}\ \mathtt{T}_2\ \mathtt{T}\ \mathtt{q}_1\ \mathtt{q}_2}$$

$$\frac{\Gamma, (\mathtt{X}:\mathtt{K}) \vdash \mathtt{e} : \mathtt{T} \triangleright \mathtt{q} \qquad \begin{array}{l} \mathtt{isAll}_{\mathtt{X,K,T}} = \mathtt{p} \\ \mathtt{stripAll}_{\mathtt{K}} = \mathtt{s} \\ \mathtt{underAll}_{\mathtt{X,K,T}} = \mathtt{u} \end{array}}{\Gamma \vdash (\Lambda \mathtt{X}:\mathtt{K}.\mathtt{e}) : (\forall \mathtt{X}:\mathtt{K}.\mathtt{T}) \triangleright \mathtt{tabs}\ (\forall \mathtt{X}:\mathtt{K}.\mathtt{T})\ \mathtt{p\ s\ u}\ (\Lambda \mathtt{X}:\mathtt{K}.\mathtt{q})}$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{e} : (\forall \mathtt{X}:\mathtt{K}.\mathtt{T}) \triangleright \mathtt{q} & \mathtt{isAll}_{\mathtt{X,K,T}} = \mathtt{p} \\ \Gamma \vdash \mathtt{A} : \mathtt{K} & \mathtt{inst}_{\mathtt{X,K,T,A}} = \mathtt{i} \end{array}}{\Gamma \vdash \mathtt{e\ A} : \mathtt{T[X{:=}A]} \triangleright \mathtt{tapp}\ (\forall \mathtt{X}:\mathtt{K}.\mathtt{T})\ (\mathtt{T[X{:=}A]})\ \mathtt{p\ i}}$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{F} : (* \to *) \to * \to * & \\ \Gamma \vdash \mathtt{T} : * & \Gamma \vdash \mathtt{e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \triangleright \mathtt{q} \end{array}}{\Gamma \vdash \mathtt{fold\ F\ T\ e} : \mu\ \mathtt{F\ T} \triangleright \mathtt{fld\ F\ T\ q}}$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{F} : (* \to *) \to * \to * & \\ \Gamma \vdash \mathtt{T} : * & \Gamma \vdash \mathtt{e} : \mu\ \mathtt{F\ T} \triangleright \mathtt{q} \end{array}}{\Gamma \vdash \mathtt{unfold\ F\ T\ e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \triangleright \mathtt{unfld\ F\ T\ q}}$$

$$\frac{\langle\rangle \vdash \mathtt{e} : \mathtt{T} \triangleright \mathtt{q}}{\begin{array}{l} \overline{\mathtt{e}} = \Lambda \mathtt{V}:* \to *. \\ \quad \lambda \mathtt{abs} : \mathtt{Abs\ V}.\ \lambda \mathtt{app} : \mathtt{App\ V}. \\ \quad \lambda \mathtt{tabs} : \mathtt{TAbs\ V}.\ \lambda \mathtt{tapp} : \mathtt{TApp\ V}. \\ \quad \lambda \mathtt{fld} : \mathtt{Fold\ V}.\ \lambda \mathtt{unfld} : \mathtt{Unfold\ V}. \\ \quad \mathtt{q} \end{array}}$$

Quotation and pre-quotation

Figure 4.5: Tagless-final self-representation of $F_\omega^{\mu i}$.

```
decl Id : * → * = λA:*. A;

decl unAbs  : Abs Id  =
  ΛA:*. ΛB:*. λf : A → B. f;
decl unApp  : App Id  =
  ΛA:*. ΛB:*. λf : A → B. f;
decl unTAbs : TAbs Id =
  ΛA:*. λp:IsAll A. λs:StripAll A. λu:UnderAll A. λe:All Id Id A.
  unAll A p Id Id e
decl unTApp : TApp Id =
  ΛA:*. ΛB:*. λp : IsAll A. λi : Inst A B. λx : A.
  i Id (sym (All Id Id A) A (unAll A p Id) Id x)
decl unFold : Fold Id =
  ΛF: (* → *) → * → *. ΛA:*. λx : F (μ F) A. fold F A x;
decl unUnfold : Unfold Id =
  ΛF: (* → *) → * → *. ΛA:*. λx : μ F A. unfold F A x;

decl unquote : (∀A:*. Exp A → A) =
  ΛA:*. λe:Exp A.
  e Id unAbs unApp unTAbs unTApp unFold unUnfold
```

Figure 4.6: The self-interpreter `unquote`.

Figure 2.4. The case function `unAbs` for λ-abstractions is unchanged. The case function `unApp` for applications is an η-contracted version of the one for F_ω. As we will discuss in Section 4.5, this helps achieve Jones-optimality. The case function `unTAbs` for type abstractions is different, due to the switch from type representations used in Chapter 2 to intensional type functions from Chapter 3. In this version, the term `e` has a type `All Id Id A`, which `unAll` coerces to `A`. It can do this because `p` proves that `A` is a quantified type of the form `∀X:K.T`. Then `All Id Id A = All Id Id (∀X:K.T)` ≡ `Id (∀X:K. Id T)` ≡ `(∀X:K.T) = A`. The case function `unTApp` is conceptually the same: we interpret a type application using the instantiation function embedded into its representation. The difference is in how we typecheck instantiation functions: in this version instantiation functions are polymorphic. Here, `i Id` has the type `All Id Id A → B`, so we coerce `x` from the type `A` to `All Id Id A`. The case functions `unFold` and `unUnfold` are new, because F_ω does not include `fold` and `unfold` operators. We interpret them meta-circularly – i.e., they implement themselves.

4.5 Jones-optimality

A partial evaluator is Jones-optimal if it can specialize away all the computational overhead caused by self-interpretation. More precisely, specializing a self-interpreter to a program should generate a compiled program that is no slower than the original program – on *any* input.

The canonical definition is from Jones, Gomard and Sestoft [53, pg. 139]. We restate it for out context of λ-calculus and using our notation:

Definition 4.5.1 (Jones-optimality)**.** *A partial evaluator* `mix` *is Jones-optimal with respect to* $time(-)$ *if for any terms* $\mathtt{p} : \mathtt{A} \to \mathtt{B}$ *and* $\mathtt{d} : \mathtt{A}$ *and for some self-interpreter* `unquote`*,* $time(\mathtt{p'}\ \mathtt{d}) \leq time(\mathtt{p}\ \mathtt{d})$*, where* $\mathtt{mix}\ \overline{\mathtt{unquote}}\ \overline{\overline{\mathtt{p}}} \equiv_\beta \overline{\mathtt{p'}}$*.*

In terms of the Futamura projections, an optimal partial evaluator should generate a compiler that never causes a slowdown.

It is important to note that in the definition of Jones optimality, both $time(-)$ and `unquote` are existentially quantified. This provides important flexibility in the definition, because different notions of $time(-)$ may be most appropriate for different settings, and because there may be multiple different implementations of `unquote` (and multiple self-representation schemes) for a single language. If `mix` were required to be able to specialize away the overhead of *any* self-interpreter, it would be very difficult to establish optimality. However, this flexibility can be abused. A partial evaluator can cheat by checking if its first input (the program being specialized) is a particular self-interpreter. If so, it can simply return its second input. Our partial evaluator is honest; it plays no such tricks.

A definition of $time(-)$ should reflect the time complexity of running a program in a real-world environment. Definitions that meet this criterion include measuring reduction steps, or measuring seconds of interpreted or compiled code. The definition we use measures β-reduction steps to normal form under lazy evaluation (i.e., normal-order reduction with memoization). This has several advantages over measuring seconds: it enables a direct comparison with Mogensen's work, it is deterministic and independent of the computer used for benchmarking, and it corresponds to how functional programming languages are actually implemented.

Specialization by normalization is not optimal for this definition of $time(-)$. For instance, compilation causes a slowdown for the factorial function for Church numerals. Not only compilation, but specialization in general can cause slowdown. This can be demonstrated by the `power` function for Church-numerals: evaluating `power three five` is faster than first normalizing `power three` and then applying the result to `five`.

The reason for these slowdowns is that lazy evaluation has a notion of *sharing*, which makes computation more efficient. Normalization can prevent sharing, which causes a slowdown. Consider a redex `(λx.a) b`. The bound variable `x` can occur zero or more times in `a`. How many times will `b` be evaluated? This depends on the evaluation strategy. Under call-by-name, the redex is evaluated by reducing `(λx.a) b` to `a[x:=b]` and then evaluating the reduct. This may involve evaluating `b` multiple times, depending on where and how many times `x` occurs in `a`. Under call-by-value, we first evaluate `b` to its value `v`, then we reduce `(λx.a) v` and continue evaluating. In this case, `b` is evaluated exactly once, not matter where or how many times `x` occurs in `a`. Each occurrence *shares* the value of `b`. Under lazy or call-by-need evaluation, we get the same sharing behavior as in call-by-value, but without having to evaluate `b` up front. Instead, we only evaluate `b` if we need to. Then we remember its value in case it's needed again later. Thus, under lazy evaluation `b` is evaluated at most once.

Reducing a redex `(λx.a) b` to `a[x:=b]` at specialization time may cause `b` to be evaluated multiple times at runtime, if we hadn't reduced it we'd evaluate `b` at most once. If `b` can be reduced to a value at specialization time, then we can reduce the redex without without risk of slowdown, since evaluating a value is zero-cost. However, since specialization occurs under λ-abstractions, `b` can contain free variables. Thus, it may be impossible to reduce it to a value at specialization time. In this case, reducing the redex can cause a slowdown at runtime. This is why specialization by normalization is not optimal: it blindly reduces every redex, potentially losing the sharing benefits provided by the evaluation strategy.

Our solution is for the partial evaluator to analyze which redexes can be safely reduced at specialization time without losing sharing at runtime. We call such redexes "specialization-safe" because they do not risk causing a slowdown in the residual code.

We now make the notion of a specialization-safe β-redex precise. We use substitutions to quantify over possible instantiations of free variables at runtime. We characterize these instantiations using substitutions that map variables to values to model sharing behavior, particularly for free variables.

Definition 4.5.2. *A value substitution is a substitution that maps variables to values.*

Definition 4.5.3. *A β-redex* `(λx.a) b` *is specialization-safe iff* $time(\theta($`a[x:=b]`$)) \leq time(\theta($`(λx.a) b`$))$ *for any value substitution θ.*

Specialization-safety is a local property. It only considers one step of β-reduction at a given position in the term. It could be generalized to longer sequences of β-reductions, possible in different positions. It could also be generalized to other kinds of transformations. However, this definition is sufficient for achieving Jones-optimality. We simplify a bit further by using a conservative condition that implies specialization safety.

Theorem 4.5.1. *A β-redex* `(λx.a) b` *is specialization-safe if either of the following is true:*

1. *$erase($`b`$)$ is not an application.*
2. `x` *occurs at most once in* `a`*, and does not occur under a nested λ-abstraction.*

Proof. Sketch: For (1), $erase($`b`$)$ must be a variable or a λ-abstraction, and so can be safely inlined. In each case, $erase(\theta($`b`$))$ is a value so its evaluation is zero-cost. For (2), the value of $erase(\theta($`b`$))$ is needed at most once, so sharing is not needed. □

These conditions are conservative: there are specialization-safe redexes for which neither (1) or (2) is true. However, we show that these conditions are sufficient for a Jones-optimal partial evaluator. This partial evaluator is also based on specialization by evaluation, using a new self-evaluator `opt` that reduces all specialization-safe redexes and no others. The result is the first typed, self-applicable partial evaluator that is Jones-optimal, never causes slowdown, and can generate the Futamura projections. The partial evaluator has the type (‡) discussed in section 4.2. The Futamura projections have the types listed in Figure 4.1 and all normalize and meet their specifications.

In general, `opt` will make programs run faster, and is guaranteed to never cause a slowdown. It is the first typed self-applicable self-optimizer, and can be useful for applications outside of partial evaluation.

We prove that our specialization-safe partial-evaluator is Jones-optimal for the version of `unquote` in Figure 4.6. Intuitively, this is true because all β-redexes related to interpretation are specialization-safe. If there are no other specialization-safe β-redexes, then `mix` $\overline{\texttt{unquote}}$ $\overline{\overline{\texttt{e}}}$ $\equiv_\beta$ $\overline{\texttt{e}}$. If there are some more specialization-safe β-redexes, then specialization-safety implies that reducing them will not cause a slowdown.

Lemma 4.5.1. *For any prequotation* $\Gamma \vdash$ `e : T` $\triangleright$ `q`*,* `q [abs:=unAbs, app:=unApp, tabs:=unTAbs, tapp:=unTApp, fld:=unFold, unfld:=unUnfold]` *reduces to* `e` *by reducing only specialization-safe redexes.*

Proof. By induction on the derivation of $\Gamma \vdash$ `e : T` $\triangleright$ `q`.

Define `θ(q) = q [abs:=unAbs, app:=unApp, tabs:=unTAbs, tapp:=unTApp, fld:=unFold, unfld:=unUnfold]`.

If `e` is a variable `x`, then `θ(q) = x = e`, and the result follows with 0 reduction steps.

If `e` is a λ-abstraction `λx:T1.e1`, then `q = abs T1 T2 (λx: T1. q1)` and Γ`,(x:T1)` $\vdash$ `e1 : T2` $\triangleright$ `q1`. By induction, `θ(q1)` safely reduces to `e1`. Now `θ(q) = unAbs T1 T2 (λx:T1. θ(q1))`, which safely reduces to `unAbs T1 T2 (λx:T1. e1)`. The abstraction over `f` in `unAbs` is safe, so `unAbs T1 T2 (λx:T1. e1)` safely reduces to `(λx:T1. e1) = e`.

If `e` is an application `e1 e2`, then `q = app T2 T q1 q2` and $\Gamma \vdash$ `e1 : T2` $\to$ `T` $\triangleright$ `q1` and $\Gamma \vdash$ `e2 : T2` $\triangleright$ `q2`. By induction, `θ(q1)` safely reduces to `e1` and `θ(q2)` safely reduces to `e2`. Now `θ(q) = unApp T2 T θ(q1) θ(q2)`, which safely reduces to `unApp T2 T e1 e2`. Since the abstraction over `f` in `unApp` is safe, `unApp T2 T e1 e2` safely reduces to `e1 e2 = e`.

If `e` is a type abstraction `(`Λ`X:K.e)`, then `T = (`$\forall$`X:K. T1)` and `q = tabs (`$\forall$`X:K.T1) isAll`$_{X,K,T1}$ `stripAll`$_K$ `underAll`$_{X,K,T1}$ `(`Λ`X:K.q1)`, and Γ`,(X:K)` $\vdash$ `e1 : T1` $\triangleright$ `q1`. By induction, `θ(q1)` safely reduces to `e1`. Now `θ(q) = unTAbs (`$\forall$`X:K.T1) isAll`$_{X,K,T1}$ `stripAll`$_K$ `underAll`$_{X,K,T1}$ `(`Λ`X:K.θ(q1))`, which safely reduces to `unTAbs (`$\forall$`X:K.T1) isAll`$_{X,K,T1}$ `stripAll`$_K$ `underAll`$_{X,K,T1}$ `(`Λ`X:K.e1)`. Since `isAll`$_{X,K,T1}$, `stripAll`$_K$, and `underAll`$_{X,K,T1}$ are values, and the abstraction over `e` in `unTAbs` is safe, `unTAbs (`$\forall$`X:K.T1) isAll`$_{X,K,T1}$ `stripAll`$_K$ `underAll`$_{X,K,T1}$ `(`Λ`X:K.e1)` safely reduces in a few steps to `unAll (`$\forall$`X:K.T1) isAll`$_{X,K,T1}$ `Id Id (`Λ`X:K.e1)`, which safely reduces to `unAll`$_{X,K,T1}$ `Id Id (`Λ`X:K.e1)`, and then to `refl (`$\forall$`X:K.T1) Id (`Λ`X:K.e1)`, and then to `(`Λ`X:K.e1) = e`.

The case for type applications is similar to the previous case.

If `e` is a `fold` expression `fold F A e1`, then `T = μ F A` and `q = fld F A q1`, and $\Gamma \vdash$ `e1 : F (μ F) A` $\triangleright$ `q1`. Now `θ(q) = unFold F A θ(q1)`, which safely reduces to `unFold F A e1`. Since the abstraction over `x` in `unFold` is safe, `unFold F A e1` safely reduces to `fold F A e1 = e`.

The case for `unfold` is similar to the previous case.

□

Lemma 4.5.2. *For any* $\langle\rangle \vdash$ `e : T`*,* `unquote T` $\overline{\texttt{e}}$ *reduces to* `e` *by reducing only specialization-safe redexes.*

Proof. First, the abstraction over `e` in `unquote` is safe, so `unquote T` $\overline{\texttt{e}}$ safely reduces to $\overline{\texttt{e}}$ `Id unAbs unApp unTAbs unTApp unFold unUnfold`. Next, each of the case functions `unAbs`, `unApp`, etc. are values, so $\overline{\texttt{e}}$ `Id unAbs unApp unTAbs unTApp unFold unUnfold` safely reduces to `q [abs:=unAbs, app:=unApp, tabs:=unTAbs, tapp:=unTApp, fld:=unFold, unfld:=unUnfold]` where $\langle\rangle \vdash$ `e : T` $\triangleright$ `q`. By Lemma 4.5.1, `q [abs:=unAbs, app:=unApp, tabs:=unTAbs, tapp:=unTApp, fld:=unFold, unfld:=unUnfold]` safely reduces to `e`. □

Theorem 4.5.2. `mix` *is Jones-optimal.*

Proof. Follows from Lemma 4.5.2. □

We mentioned in Section 4.4 that the case function `unApp` in Figure 4.6 is an η-contracted version of the one from the F_ω self-interpreter in Figure 2.4. Suppose we defined `unApp` to be the η-expanded version, `unApp = ΛA:*. ΛB:*. λf : A → B. λx : A. f x`. Since the variable `f` is referenced from within the λ-abstraction over `x`, `unApp T1 T2 e1 e2` would not be safely reducible `e1 e2`. Then our partial-evaluator would not be Jones-optimal. This highlights the sensitivity of Jones-optimality to the choice of self-interpreter. Jones-optimality is also sensitive to the definition of $time(-)$, which includes which evaluation strategy is used at runtime. Specialization by normalization is optimal for call-by-name, since it provides no sharing. Our specialization-safe partial evaluator is optimal for call-by-name, call-by-value, and call-by-need evaluation.

4.6 Implementation

Figure 4.7 shows the highlights of our Jones-optimal partial evaluator. At its core is the self-evaluator `opt`, which is a modification of the normalization-by-evaluation algorithm that only reduces specialization-safe redexes. The name `opt` underscores that it optimizes its input program, and never causes slowdown. The function `sem` is a fold that converts an `Exp A` to a semantic object of type `Sem V A`, and `reify` extracts a representation of the evaluated term from the semantic object. The partial evaluator `mix` is similar to the one shown in Figure 4.4: it optimizes rather than normalizes the application of its two inputs. A notable difference is the use of `forceExp`. This is an important optimization for self-application, specifically when specializing `mix` to a single input. It checks that both arguments are available before optimizing, thereby preventing a blow up in the output code size from many copies of `opt`'s traversal functions being residualized thoughout `f`.

The type `Sem` of semantic objects is shown in Figure 4.8. It is a recursive intensional type function like the type `CPS` from Figure 3.16. The type `Sem V A` is equivalent to `μ (SemF V) A` and isomorphic (via (un)folding the recursion) to `SemF (Sem V) A`. `SemF` defines the pattern-matching interface for semantic objects.

There are two variants of semantic object: neutral objects that do not form redexes (statically-safe or otherwise) when in head-position, and non-neutral objects that do. Neutral objects contain a boolean and a representation. The representation is used to `reify` the semantic object. The boolean is `true` if that representation is of a term that erases to a variable. If so, then any redex in which this term is in argument position is statically-safe.

Active objects form redexes when in head-position. Their first two components are similar to those of neutral objects. The boolean is `true` is the representation erases to a contain a boolean. The third component is used to compute the reduct of redexes with this object in head-position. These have types of the form `SemF1 (Sem V) A`. `SemF1` is an intensional type function that depends on the structure of `A`.

$$\begin{aligned}
\texttt{SemF1 (Sem V) (A} \to \texttt{B)} &\equiv \texttt{Pair Bool (Sem V A} \to \texttt{Sem V B)} \\
\texttt{SemF1 (Sem V) (}\forall\texttt{X:K.T)} &\equiv \forall\texttt{X:K. Sem V T} \\
\texttt{SemF1 (Sem V) (}\mu\texttt{ F A)} &\equiv \texttt{Sem V (F (}\mu\texttt{ F) A)}
\end{aligned}$$

A non-neutral semantic object of arrow type is a λ-abstraction. For these, the third component is a pair of boolean and function from semantic objects to semantic objects. The function computes semantic objects for reducts of this abstraction. The boolean indicates whether the λ-abstraction is statically safe according to condition 2 of 4.5.1.

If a non-neutral semantic object has a quantified type, then it is a Λ-abstraction. For these, the third component is a polymorphic semantic object. If it has a recursive type, it is for a `fold` term `fold F A e`, and the third component is the semantic object of the unfolded term `e`.

Figure 4.9 shows the key functions used by `sem` to reduce function applications. Given semantic objects for a function `f` and an argument `x`, `semApp` first does case analysis on `f`. If it's neutral, then the application is not a redex and is residualized. Otherwise, the appication is a redex. To check if it's statically safe, we test whether either condition 1 or 2 of Theorem 4.5.1 are true. We reduce if so, and otherwise we residualize. Whenever we residualize, we use `semNe` to construct a semantic object for the residual term. The argument `false` to `semNe` indicates that result is an application.

```
sem      : (∀V:* → *. ∀A : *. Exp A → Sem V A)
reify    : (∀V:* → *. ∀A : *. Sem V A → PExp V A)
forceExp : (∀T:*. Exp T → (∀A:*. A → A))

decl opt : (∀A : *. Exp A → Exp A) =
  ΛA:*. λe : Exp A. ΛV:* → *.
  reify V A (sem V A e);

decl mix : (∀A : *. ∀B : *. Exp (A → B) → Exp A → Exp B) =
  ΛA:*.ΛB:*. λf:Exp (A → B). λx:Exp A.
  let e : Exp B = (ΛV:* → *. app V A B (f V) (x V)) in
  opt B (forceExp A x (Exp B) e);
```

Figure 4.7: Highlights of our Jones-optimal `mix`

```
decl SemF1 : (∗ → ∗) → ∗ → ∗ =
  λSem : ∗ → ∗. λA:∗.
  Typecase
    (λA1:∗. λA2:∗. Pair Bool (Sem A1 → Sem A2))
    Id Sem
    (λF : (∗ → ∗) → ∗ → ∗. λB : ∗.
     Sem (F (μ F) B))
    A

decl SemF : (∗ → ∗) → (∗ → ∗) → ∗ → ∗ =
  λV : ∗ → ∗. λSem : ∗ → ∗. λA : ∗.
  ∀R:∗.
  (Bool → PExp V A → R) →                      -- Neutral
  (Bool → PExp V A → SemF1 Sem A → R) →        -- Non-neutral
  R

decl Sem : (∗ → ∗) → ∗ → ∗ = λV : ∗ → ∗. μ (SemF V)
```

Figure 4.8: Types of our Jones-optimal partial evaluator.

```
decl semApp:(∀V:*→*. ∀B:*. ∀A:*. Sem V (B→A) → Sem V B → Sem V A) =
  ΛV:* → *. ΛB:*. ΛA:*. λf:Sem V (B → A). λx:Sem V B.
  unfold (SemF V) (B → A) f
    (Sem V A)
    -- Neutral
    (λ:Bool. λf: (PExp V (B → A)).
     semNe V A false (app V B A f (reify V B x)))
    -- Non-neutral
    (λ:Bool. λrep_f:PExp V (B → A). λp:Pair Bool (Sem V B → Sem V A).
     let cond1:Bool = notApp V B x in
     let cond2:Bool = fst Bool (Sem V B → Sem V A) p in
     or cond1 cond2
       (Sem V A)
       -- safe: reduce
       (snd Bool (Sem V B → Sem V A) p x)
       -- not safe: residualize
       (semNe V A false (app V B A rep_f (reify V B x))))
```

Figure 4.9: A key function from `sem`.

4.7 Experiments

We measured speedups from our partial evaluators using a modified version of Haskell implementation of $F_\omega^{\mu i}$ that was discussed in Section 3.7. Modifications include changes to the quoter to produce Böhm-Berarducci encoded representations, and changes to the evaluator to count reduction steps. We verified that the self-interpreter, partial-evaluators, and four Futamura projections all type check in $F_\omega^{\mu i}$.

We conducted two kinds of experiment: first, we use our β-equivalence checker to verify that the four Futamura projections satisfy their correctness specification (Figure 4.2). Second, we use measure the speedups obtained by our partial evaluators.

Following Mogensen, we measure the number of β-reduction steps needed to normalize a term using "strong" lazy evaluation. Ordinary lazy evaluation is weak – it does not reduce under λ-abstractions. Therefore, it does not always evaluate to a β-normal form. "Strong" lazy evaluation is like normal-order reduction in that it reduces to normal form, and like ordinary lazy evaluation in that it memoizes the values of variables.

Table 4.2 shows the speedups obtained by specializing a variety of subject programs. Each subject program consists of a function applied to two or more inputs. The Baseline Steps is the number of β-reduction steps to reduce that program to normal form. The Specialized Steps is the number of steps for the same program after the function has been specialized to its first input. The `power` function tests shows that specialization by normalization can cause slowdown, and the slowdown gets worse with larger exponents. The measurements show that our safe specializer is more conservative than specialization by normalization – it often gives less significant speedups, but never causes slowdown.

Table 4.3 empirically test Jones-optimality. It reports the speedups obtained by compilation via the first Futamura projection. Each subject program consists of a function applied to one or more inputs. The Baseline Steps is the number of β-reduction steps to reduce that program to normal form. The Compiled Steps is the number of steps for same program after the function has been compiled using the first Futamura projection. We see exact 1.0 compilation speedups for the `Ackermann` function, which indicate the original and compiled versions have the same performance for those inputs. This is because `Ackermann` as written is a normal form, so it compiles to itself. The `cube` and `fact` functions demonstrate non-optimality of specialization by normalization. Compilation causes a slowdown of each program for inputs greater than 3. The slowdowns get progressively worse with larger numbers. Specialization by optimization is Jones-optimal: compilation never causes a slowdown. The worst case is a 1.00 speedup – which means no speedup. For large inputs to `cube` and `fact`, the speedup tends to 1.00.

Specialization by optimization is also Jones-optimal for call-by-value, and specialization by normalization is not. For call-by-name, both are Jones-optimal but specialization by normalization provides better speedups; all redexes are specialization safe for call-by-name. Measurements for call-by-value and call-by-need are shown in Tables 4.4–4.7. Of the three evaluation strategies, call-by-name is the least efficient and least widely used. As an extreme example, computing the third Futamura projection for our specialization by optimization partial evaluator takes $\approx 10^6$ steps under lazy evaluation, $\approx 10^7$ steps under call-by-value, and $\approx 10^{555}$ steps under call-by-name. In fact, due to this inefficiency we were only able to make these measurements using a modified version lazy evaluation, in which we memoized

Program	**Specializer**	**Baseline Steps**	**Specialized Steps**	**Speedup**
`Ackermann 0 3`	normalize	7	3	2.33
	optimize	7	3	2.33
`Ackermann 1 3`	normalize	19	13	1.46
	optimize	19	14	1.36
`Ackermann 2 3`	normalize	76	65	1.17
	optimize	76	69	1.10
`Ackermann 3 3`	normalize	3685	3621	1.02
	optimize	3685	3673	1.00
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{0 3}$	normalize	58	7	8.29
	optimize	58	7	8.29
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{1 3}$	normalize	175	19	9.21
	optimize	175	19	9.21
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{2 3}$	normalize	749	76	9.86
	optimize	749	76	9.86
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{3 3}$	normalize	36027	3685	9.78
	optimize	36027	3685	9.78
$\texttt{mix}\ \overline{\texttt{unquote}}\ \overline{\overline{\texttt{Ackermann}}}$	normalize	3439	1562	2.20
	optimize	32934	26073	1.26
$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{unquote}}}$	normalize	377029	113546	3.32
	optimize	377187	317752	1.19
$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{mix}}}$	normalize	18762389	5423355	3.46
	optimize	931715	739797	1.26
`power 2 2`	normalize	145	81	1.79
	optimize	145	82	1.77
`power 3 2`	normalize	251	217	1.16
	optimize	251	153	1.64
`power 4 2`	normalize	413	533	**0.77**
	optimize	413	272	1.52
`power 5 2`	normalize	687	1253	**0.55**
	optimize	687	487	1.41
`power 6 2`	normalize	1185	2869	**0.41**
	optimize	1185	894	1.33
`power 7 2`	normalize	2131	6453	**0.33**
	optimize	2131	1685	1.26

Table 4.2: Speedups from our two partial evaluators under lazy evaluation

Program	**Compiler**	**Baseline Steps**	**Compiled Steps**	**Speedup**
`Ackermann 0 3`	normalize	7	7	1.00
	optimize	7	7	1.00
`Ackermann 1 3`	normalize	19	19	1.00
	optimize	19	19	1.00
`Ackermann 2 3`	normalize	76	76	1.00
	optimize	76	76	1.00
`Ackermann 3 3`	normalize	3685	3685	1.00
	optimize	3685	3685	1.00
`cube 1`	normalize	56	31	1.81
	optimize	56	33	1.70
`cube 2`	normalize	135	135	1.00
	optimize	135	103	1.31
`cube 3`	normalize	290	369	**0.79**
	optimize	290	249	1.16
`cube 4`	normalize	551	787	**0.70**
	optimize	551	501	1.10
`cube 5`	normalize	948	1443	**0.66**
	optimize	948	889	1.07
`fact 1`	normalize	42	20	2.10
	optimize	42	27	1.56
`fact 2`	normalize	90	67	1.34
	optimize	90	64	1.41
`fact 3`	normalize	192	263	**0.73**
	optimize	192	153	1.25
`fact 4`	normalize	522	1261	**0.41**
	optimize	522	468	1.12
`fact 5`	normalize	2064	7295	**0.28**
	optimize	2064	1993	1.04

Table 4.3: Speedups from our generated compilers under lazy evaluation

Program	**Specializer**	**Baseline Steps**	**Specialized Steps**	**Speedup**
`Ackermann 0 3`	normalize	7	3	2.33
	optimize	7	3	2.33
`Ackermann 1 3`	normalize	19	13	1.46
	optimize	19	14	1.36
`Ackermann 2 3`	normalize	76	65	1.17
	optimize	76	69	1.10
`Ackermann 3 3`	normalize	3685	3621	1.02
	optimize	3685	3673	1.00
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{0 3}$	normalize	59	7	8.43
	optimize	59	7	8.43
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{1 3}$	normalize	175	19	9.21
	optimize	175	19	9.21
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{2 3}$	normalize	749	76	9.86
	optimize	749	76	9.86
$\texttt{unquote}\ \overline{\texttt{Ackermann}}\ \texttt{3 3}$	normalize	36027	3685	9.78
	optimize	36027	3685	9.78
$\texttt{mix}\ \overline{\texttt{unquote}}\ \overline{\overline{\texttt{Ackermann}}}$	normalize	5527	1705	3.24
	optimize	56976	42473	1.34
$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{unquote}}}$	normalize	551602	124648	4.43
	optimize	1045746	860095	1.22
$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{mix}}}$	normalize	27989802	5792957	4.83
	optimize	8717337	6990449	1.25
`power 2 2`	normalize	174	139	1.25
	optimize	174	98	1.78
`power 3 2`	normalize	292	588	**0.50**
	optimize	292	177	1.65
`power 4 2`	normalize	466	2315	**0.20**
	optimize	466	304	1.53
`power 5 2`	normalize	752	8862	**0.08**
	optimize	752	527	1.43
`power 6 2`	normalize	1262	33499	**0.04**
	optimize	1262	942	1.34
`power 7 2`	normalize	2220	125852	**0.02**
	optimize	2220	1741	1.28

Table 4.4: Speedups from our two partial evaluators under call-by-value

Program	**Compiler**	**Baseline Steps**	**Compiled Steps**	**Speedup**
`Ackermann 0 3`	normalize	7	7	1.00
	optimize	7	7	1.00
`Ackermann 1 3`	normalize	19	19	1.00
	optimize	19	19	1.00
`Ackermann 2 3`	normalize	76	76	1.00
	optimize	76	76	1.00
`Ackermann 3 3`	normalize	3685	3685	1.00
	optimize	3685	3685	1.00
`cube 1`	normalize	56	31	1.81
	optimize	56	33	1.70
`cube 2`	normalize	135	135	1.00
	optimize	135	103	1.31
`cube 3`	normalize	290	369	**0.79**
	optimize	290	249	1.16
`cube 4`	normalize	551	787	**0.70**
	optimize	551	501	1.10
`cube 5`	normalize	948	1443	**0.66**
	optimize	948	889	1.07
`fact 1`	normalize	42	20	2.10
	optimize	42	27	1.56
`fact 2`	normalize	90	78	1.15
	optimize	90	64	1.41
`fact 3`	normalize	192	369	**0.52**
	optimize	192	153	1.25
`fact 4`	normalize	522	2088	**0.25**
	optimize	522	468	1.12
`fact 5`	normalize	2064	13783	**0.15**
	optimize	2064	1993	1.04

Table 4.5: Speedups from our generated compilers under call-by-value

Program	**Specializer**	**Baseline Steps**	**Specialized Steps**	**Speedup**
`Ackermann 0 3`	normalize	7	3	2.33
	optimize	7	3	2.33
`Ackermann 1 3`	normalize	19	13	1.46
	optimize	19	14	1.36
`Ackermann 2 3`	normalize	79	65	1.22
	optimize	79	69	1.14
`Ackermann 3 3`	normalize	3739	3621	1.03
	optimize	3739	3673	1.02
`unquote` $\overline{\texttt{Ackermann}}$ `0 3`	normalize	58	7	8.29
	optimize	58	7	8.29
`unquote` $\overline{\texttt{Ackermann}}$ `1 3`	normalize	178	19	9.37
	optimize	178	19	9.37
`unquote` $\overline{\texttt{Ackermann}}$ `2 3`	normalize	778	79	9.85
	optimize	778	79	9.85
`unquote` $\overline{\texttt{Ackermann}}$ `3 3`	normalize	37378	3739	10.00
	optimize	37378	3739	10.00
`mix` $\overline{\texttt{unquote}}$ $\overline{\overline{\texttt{Ackermann}}}$	normalize	4057	1562	2.60
	optimize	2.36×10^{16}	1.74×10^{16}	1.35
`mix` $\overline{\texttt{mix}}$ $\overline{\overline{\texttt{unquote}}}$	normalize	470270	125285	3.75
	optimize	4.14×10^{34}	3.02×10^{34}	1.37
`mix` $\overline{\texttt{mix}}$ $\overline{\overline{\texttt{mix}}}$	normalize	23946795	6045551	3.96
	optimize	2.01×10^{555}	1.50×10^{555}	1.35
`power 2 2`	normalize	256	81	3.16
	optimize	256	116	2.21
`power 3 2`	normalize	724	240	3.02
	optimize	724	336	2.15
`power 4 2`	normalize	1899	659	2.88
	optimize	1899	905	2.10
`power 5 2`	normalize	4833	1744	2.77
	optimize	4833	2359	2.05
`power 6 2`	normalize	12108	4509	2.69
	optimize	12108	6028	2.01
`power 7 2`	normalize	30056	11476	2.62
	optimize	30056	15204	1.98

Table 4.6: Speedups from our two partial evaluators under call-by-name

Program	**Compiler**	**Baseline Steps**	**Compiled Steps**	**Speedup**
`Ackermann 0 3`	normalize	7	7	1.00
	optimize	7	7	1.00
`Ackermann 1 3`	normalize	19	19	1.00
	optimize	19	19	1.00
`Ackermann 2 3`	normalize	79	79	1.00
	optimize	79	79	1.00
`Ackermann 3 3`	normalize	3739	3739	1.00
	optimize	3739	3739	1.00
`cube 1`	normalize	58	31	1.87
	optimize	58	33	1.76
`cube 2`	normalize	235	135	1.74
	optimize	235	141	1.67
`cube 3`	normalize	638	369	1.73
	optimize	638	381	1.67
`cube 4`	normalize	1363	787	1.73
	optimize	1363	807	1.69
`cube 5`	normalize	2506	1443	1.74
	optimize	2506	1473	1.70
`fact 1`	normalize	44	20	2.20
	optimize	44	27	1.63
`fact 2`	normalize	142	69	2.06
	optimize	142	94	1.51
`fact 3`	normalize	526	275	1.91
	optimize	526	365	1.44
`fact 4`	normalize	2364	1321	1.79
	optimize	2364	1700	1.39
`fact 5`	normalize	12876	7615	1.69
	optimize	12876	9533	1.35

Table 4.7: Speedups from our generated compilers under call-by-name

the cost of evaluating each variable in addition to its value.

4.8 Related Work

The following table compares our results with previous work in three dimensions: whether the partial evaluator operates on a typed representation, whether it is self-applicable and generates the Futamura projections, and whether it is Jones-optimal.

	Typed representation	Futamura projections	Jones optimal
Original mix [55]		√	
Lambda-mix [47]		√	
Similix [15]		√	√
Schism [32]		√	
Untyped SbN [62]		√	
TDPE [36]		√	
Carette et al. [22]	√		
Typed SbN	√	√	
Typed SbO	√	√	√

Typed representation. Typed representation and typed meta-programming guarantee strong correctness properties: they prevent type errors in generated code, and ensure that meta-programs have the intended effect on the type of a program. Carette et al. [22] implemented a typed partial evaluator that operated on a typed tagless-final representation. Their object language was the simply-typed λ-calculus extended with integers, booleans, and a fixpoint operator, and their meta-languages was MetaOCaml. Their partial evaluator could not generate the Futamura projections and was not Jones-optimal.

Untyped and simply-typed Futamura Projections. A variety of partial evaluators for untyped languages have enabled the three classical Futamura projections including the original mix [55], Lambda-mix [47], Similix [15], Schism [32], and Mogensen's untyped specialization by normalization [62]. Gluck [46, 45] popularized the fourth Futamura projection and showed how to achieve it.

The classical approach to meta-programming in statically-typed languages is to use an untyped representation, which assigns a single type to all program representations. Launchbury [58] described a simply-typed partial evaluator for LML, which enables the first and second Futamura projections.

Danvy [36] presented type-directed partial evaluation for a simply-typed language and showed how it enables the first and second Futamura projections.

Our partial evaluator is polymorphically-typed and enables all the Futamura projections. Our work differs from previous work in that the polymorphic type of our partial evaluator lets type checking rule out a larger class of bugs.

Jones Optimality. Similix [15] achieved Jones optimality for untyped language, and work by Taha, Makholm, and Hughes [83], and by Danvy and Lopez [37], achieved Jones optimality for simply-typed languages. Gade and Glück [42] presented the first mathematical proof of Jones optimality.

Our proof of Jones-optimality is different from that of Gade and Glück. They separate out the definition of $time(-)$ in two steps: first they show that the optimizations performed by the partial evaluator monotonically decrease $time(-)$. Then they define a set of programs that cannot be "more optimized" by the specializer. To prove a specializer is Jones-optimal, they show that compilation via the first Futamura projection is the identity on programs in that set. We could use a similar approach for our proof of Jones-optimality. The set would consist of the outputs of `opt` – that is, terms that have no statically-safe β-redexes, and no redexes formed by type-abstraction and type-application or by `fold` and `unfold`.

Terminating Partial Evaluation. Asai et al. [8] proved various correctness properties of a partial evaluator, including termination. The partial evaluator is implemented in a language that is more expressive that the language being partially evaluated, so self-application is impossible. Our partial evaluator does not always terminate, though our experiments demonstrate termination for important cases.

Other Related Work. Bondorf and Dussart [16], Birkedal and Welinder [14], and Thiemann [85] have shown how to write strikingly simple compiler generators by hand for untyped or simply-typed languages. This is a major alternative to the use of a partial evaluator to produce a compiler generator via the third Futamura projection. However, the task to write a polymorphically-typed compiler generator by hand is significantly harder. Open question: can one write a polymorphically-typed compiler generator by hand that is simpler than the one we have generated automatically?

Techniques have been developed to prove semantic correctness of partial evaluation, including a recent paper by Hirota and Asai [51]. We experimentally validate correctness of our partial evaluators and leave a formal correctness proof for future work.

Shali and Cook [77] and Brady and Hammond [17] implemented partial evaluators for Java and for a dependently typed language, respectively. Each supports the first Futamura projection only. It is an open question whether those approaches can be extended to support all four Futamura projections.

4.9 Conclusion

The typed, self-applicable, Jones-optimal partial evaluator developed in this chapter demonstrates the practicality of typed self-applicable meta-programming. We answers several long-standing open questions: whether it's possible to program a self-applicable partial evaluator that operates on typed representations, whether such a partial evaluator can be Jones-optimal, and whether it can generate the four Futamura projections.

The key to achieving Jones-optimality and preventing slowdown from specialization is our notion of specialization-safe β-redexes. This notion can be easily generalized to other kinds of redex and other languages with term-rewriting operational semantics. Whether a simple syntactic check like ours will exist in other settings will depend on the particulars of each setting.

Our results open new challenges for future work. One limitation of our partial evaluator is that specialization doesn't always terminate. Is it possible to ensure a typed self-applicable and Jones-optimal partial evaluator terminates on all inputs? Also, that our partial evaluator is Jones-optimal does not imply that its speedups are maximal. What new techniques can be used to define a typed self-applicable and Jones-optimal partial evaluator with greater speedups than ours?

CHAPTER 5

Conclusion

This dissertation demonstrates the feasibility and usefulness of typed self-applicable meta-programming. It includes several key contributions. Chapter 2 shows that typed representation makes self-interpretation possible even for strongly normalizing languages. Potential applications for proof assistants, which are based on sound logics that correspond to strongly normalizing languages via the Curry-Howard correspondence. Typed self-representation in such a setting can be used to facilitate proofs by reflection, a powerful general-purpose proof automation technique. Within the field of mechanized meta-theory, there is a line of work that aims to formalize such logics in themselves. There are some similarities, for example, typed representation is a core technique of both typed meta-programming and mechanized meta-theory. However, the latter is more interested in proving deep meta-linguistic properties than with self-application. Indeed, Gödel's second incompleteness theorem states that one important meta-linguistic property, namely consistency of the language, cannot be proved in the language itself. Rather, the current focus is to formalize a core fragment of a logic in itself.

Chapters 3 and 4 present the first statically-typed language that supports both kinds of self-interpreter and a self-applicable partial evaluator. That the language $F_\omega^{\mu i}$ extends F_ω indicates that self-evaluation seems to be fundamentally more difficult than self-recognition. In particular, self-evaluation seems to need some kind of recursion, and a notion of type equality such that is often used to encode GADTs. While the self-evaluators `nbe` and `opt` defined in Chapter 4 for the Böhm-Berarducci self-representation defined there do not use the fixpoint combinator, they both require recursive types, specifically recursive intensional type functions.

The self-representations of $F_\omega^{\mu i}$ used in Chapter 3 Chapter 4 are isomorphic to each other, and functions to convert between them can be defined in $F_\omega^{\mu i}$ itself. Each representation has its own tradeoffs. The Mogensen-Scott encoding of Chapter 3 supports non-primitive-recursive functions more easily than the Böhm-Berarducci encoding of Chapter 4. On the other hand, primitive-recursive functions (folds) are more efficient on the Böhm-Berarducci encoding, and do not require a fixpoint combinator. If we want to support self-applicable partial evluation and all the self-evaluators from Chapter 3, all with a single canonical representation and quoter, the Böhm-Berarducci representation is the better choice. The self-evaluators that use the Mogensen-Scott representation can be easily adapted for the Böhm-Berarducci representation by composition with the conversion functions between them. On the other hand, our partial evaluators cannot be adapted for Mogensen-Scott encoding without using a fixpoint combinator, which would make the Futamura projections no longer terminate.

The techniques developed here are applicable to statically typed functional languages like Haskell or OCaml that include GADTs and type-level computation. They may also be applicable in Scala, which has a built-in notion of type equality. However, more work is required to achieve typed self-applicable meta-programming in those languages. Our techniques can be used to define new high-level languages that support typed self-applicable meta-programming. For example, such a language could desugar to $F_\omega^{\mu i}$ or an extension of it. This dissertation has solved the foundational problems of practical typed self-applicable meta-programming. Our techniques would be compatible with extensions to $F_\omega \mu i$ including sum and product types, integers, booleans, and side effects.

Bibliography

[1] The webpage accompanying this chapter is available at http://compilers.cs.ucla.edu/popl16/. The full paper with the appendix is available there, as is the source code for our implementation of System F_ω and our operations.

[2] The webpage accompanying this chapter is available at http://compilers.cs.ucla.edu/popl17/. The full paper with the appendix is available there, as is the source code for our implementation of System $F_\omega^{\mu i}$ and our operations.

[3] Abadi, M., Cardelli, L., and Plotkin, G. Types for the scott numerals, 1993.

[4] Abelson, H., and Sussman, G. *Structure and Interpretation of Computer Programs.* MIT electrical engineering and computer science series. MIT Press, 1987.

[5] Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[6] Aehlig, K., and Joachimski, F. Operational aspects of untyped normalisation by evaluation. *Mathematical Structures in Computer Science 14* (8 2004), 587–611.

[7] Altenkirch, T., and Kaposi, A. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016), POPL '16, ACM.

[8] Asai, K., Fennell, L., Thiemann, P., and Zhang, Y. A type theoretic specification of partial evaluation. In *PPDP'14 Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming* (2014), pp. 57–68.

[9] Baars, A. I., and Swierstra, S. D. Typing dynamic typing. In *ICFP '02: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming* (2002), ACM Press, pp. 157–166.

[10] Barendregt, H. Self-interpretations in lambda calculus. *J. Funct. Program 1*, 2 (1991), 229–233.

[11] Barendregt, H. *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures: Abramski, S. (ed).* Oxford University Press, Inc., New York, NY, 1993, ch. Lambda Calculi with Types.

[12] Barras, B., and Werner, B. Coq in coq. Tech. rep., 1997.

[13] Berarducci, A., and Böhm, C. A self-interpreter of lambda calculus having a normal form. In *CSL* (1992), pp. 85–99.

[14] Birkedal, L., and Welinder, M. Hand-writing program generator generators. In *Programming Language Implementation and Logic Programming* (1994), pp. 198–214.

[15] Bondorf, A., and Danvy, O. Automatic autoprojection of recursive equations with global variables and abstract data types. *Sci. Comput. Programming 16* (1991), 151–195.

[16] Bondorf, A., and Dussart, D. Improving CPS-based partial evaluation: Writing cogen by hand. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1994), pp. 1–10.

[17] Brady, E. C., and Hammond, K. Scrapping your inefficient engine: Using partial evaluation to improve domain-specific language implementation. In *Proceedings of ICFP'10, ACM SIGPLAN International Conference on Functional Programming* (2010).

[18] Braithwaite, R. The significance of the meta-circular interpreter. http://weblog.raganwald.com/2006/11/significance-of-meta-circular_22.html, November 2006.

[19] Brown, M., and Palsberg, J. Self-Representation in Girard's System U. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL '15, ACM, pp. 471–484.

[20] Brown, M., and Palsberg, J. Breaking through the normalization barrier: A self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL 2016, ACM, pp. 5–17.

[21] Brown, M., and Palsberg, J. Typed self-evaluation via intensional type functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 2017), POPL 2017, ACM, pp. 415–428.

[22] Carette, J., Kiselyov, O., and Shan, C. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming 19*, 5 (2009), 509–543.

[23] Chapman, J. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science 228* (2009), 21–36.

[24] Chen, C., and Xi, H. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden, August 2003), pp. 275–286.

[25] Chen, C., and Xi, H. Meta-Programming through Typeful Code Representation. *Journal of Functional Programming 15*, 6 (2005), 797–835.

[26] Chen, C., Zhu, D., and Xi, H. *Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 239–254.

[27] Cheney, J., and Hinze, R. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA, 2002), Haskell '02, ACM, pp. 90–104.

[28] Cheney, J., and Hinze, R. First-class phantom types. Tech. rep., Cornell University, 2003.

[29] Chlipala, A. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2008), ICFP '08, ACM, pp. 143–156.

[30] Chlipala, A. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[31] Collins, G. D., and Shao, Z. Intensional analysis of higher-kinded recursive types. Tech. rep., Yale University, 2002.

[32] Consel, C. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1993), pp. 145–154.

[33] Crary, K., and Weirich, S. Flexible type analysis. In *In 1999 ACM International Conference on Functional Programming* (1999), ACM Press, pp. 233–248.

[34] Crary, K., Weirich, S., and Morrisett, G. Intensional polymorphism in type-erasure semantics. *SIGPLAN Not. 34*, 1 (September 1998), 301–312.

[35] Cretin, J., and Rémy, D. On the power of coercion abstraction. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 361–372.

[36] Danvy, O. Type-directed partial evaluation. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (1996), pp. 242–257.

[37] Danvy, O., and López, P. E. M. Tagging, encoding, and Jones optimality. In *Proceedings of ESOP'03, European Symposium on Programming* (2003), Springer-Verlag (*L*NCS), pp. 335–347.

[38] Duggan, D. *A type-based semantics for user-defined marshalling in polymorphic languages.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 273–297.

[39] Eich, B. Narcissus. `http://mxr.mozilla.org/mozilla/source/js/narcissus/jsexec.js`, 2010.

[40] Filinski, A., and Korsholm Rohde, H. *A Denotational Account of Untyped Normalization by Evaluation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 167–181.

[41] Futamura, Y. Evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation 12* (1999), 381–391. First published in 1971 in System.Computers.Controls, Volume 2, Number 5, pages 45–50.

[42] Gade, J., and Gluck, R. On Jones-optimal specializers: A case study using unmix. In *Proceedings of APLAS'06, Asian Symposium on Programming Languages and Systems* (2006), Springer-Verlag (*L*NCS 4279), pp. 406–422.

[43] Gibbons, J., and Wu, N. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2014), ICFP '14, ACM, pp. 339–347.

[44] Girard, J.-Y., Taylor, P., and Lafont, Y. *Proofs and Types.* No. 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[45] Glück, R. An experiment with the fourth Futamura projection. In *Ershov Memorial Conference* (2009), pp. 135–150.

[46] Glück, R. Is there a fourth futamura projection? In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (New York, NY, USA, 2009), PEPM '09, ACM, pp. 51–60.

[47] Gomard, C. K., and Jones, N. D. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming 1*, 1 (1991), 21–69.

[48] Guillemette, L.-J., and Monnier, S. A type-preserving compiler in haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2008), ICFP '08, ACM, pp. 75–86.

[49] Harper, R., and Licata, D. R. Mechanizing metatheory in a logical framework. *J. Funct. Program. 17*, 4-5 (July 2007), 613–673.

[50] Harper, R., and Morrisett, G. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), POPL '95, ACM, pp. 130–141.

[51] Hirota, N., and Asai, K. Formalizing a correctness property of a type-directed partial evaluator. In *PLPV'14 Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages meets Program Verification* (2014), pp. 41–46.

[52] Jay, B., and Palsberg, J. Typed self-interpretation by pattern matching. In *Proceedings of ICFP'11, ACM SIGPLAN International Conference on Functional Programming* (Tokyo, September 2011), pp. 247–258.

[53] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[54] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[55] Jones, N. D., Sestoft, P., and Søndergaard, H. An experiment in partial evaluation: The generation of a compiler generator. In *Proceedings of Rewriting Techniques and Applications* (1985), J.-P. Jouannaud, Ed., Springer-Verlag (*L*NCS 202), pp. 225–282.

[56] Kiselyov, O. Metatypechecking: Staged typed compilation into gadt using typeclasses. http://okmij.org/ftp/tagless-final/tagless-typed.html#tc-GADT-tc.

[57] Kleene, S. C. λ-definability and recursiveness. *Duke Math. J.* (1936), 340–353.

[58] Launchbury, J. A strongly-typed self-applicable partial evaluator. In *Proceedings of FPCA'91, Sixth ACM Conference on Functional Programming Languages and Computer Architecture* (1991), pp. 145–164.

[59] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM 3*, 4 (April 1960), 184–195.

[60] Middelkoop, A., Dijkstra, A., and Swierstra, S. D. A lean specification for gadts: System f with first-class equality proofs. *Higher Order Symbol. Comput. 23*, 2 (June 2010), 145–166.

[61] Mitchell, J. C. Polymorphic type inference and containment. *Inf. Comput. 76*, 2-3 (February 1988), 211–249.

[62] Mogensen, T. A. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 1995), PEPM '95, ACM, pp. 39–44.

[63] Mogensen, T. Æ. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming 2*, 3 (1992), 345–363. See also DIKU Report D–128, Sep 2, 1994.

[64] Morrisett, G. Compiling with types. Tech. rep., 1995.

[65] Morrisett, G. F-omega – the workhorse of modern compilers. http://www.eecs.harvard.edu/ greg/cs256sp2005/lec16.txt, 2005.

[66] Naylor, M. Evaluating Haskell in Haskell. *The Monad.Reader 10* (2008), 25–33.

[67] Neis, G., Dreyer, D., and Rossberg, A. Non-parametric parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2009), ICFP '09, ACM, pp. 135–148.

[68] Pasalic, E. *The Role of Type Equality in Meta-programming.* PhD thesis, 2004. AAI3151199.

[69] Pfenning, F., and Lee, P. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science 89*, 1 (1991), 137–159.

[70] Pierce, B. C. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[71] Rendel, T., Ostermann, K., and Hofer, C. Typed self-representation. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2009), pp. 293–303.

[72] Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference* (1972), ACM Press, pp. 717–740. The paper later appeared in Higher-Order and Symbolic Computation, 11, 363–397 (1998).

[73] Rigo, A., and Pedroni, S. Pypy's approach to virtual machine construction. In *OOPSLA Companion* (2006), pp. 044–953.

[74] Rossberg, A. HaMLet. http://www.mpi-sws.org/ rossberg/hamlet, 2010.

[75] Saha, B., Trifonov, V., and Shao, Z. Intensional analysis of quantified types. *ACM Trans. Program. Lang. Syst. 25*, 2 (2003), 159–209.

[76] Schürmann, C., Yu, D., and Ni, Z. A representation of f_ω in lf. *Electronic Notes in Theoretical Computer Science 58*, 1 (2001), 79 – 96.

[77] Shali, A., and Cook, W. Hybrid partial evaluation. In *Proceedings of OOPSLA'11, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (2011), pp. 375–390.

[78] Sheard, T., and Jones, S. P. Template meta-programming for haskell. *SIGPLAN Not. 37*, 12 (December 2002), 60–75.

[79] Sheard, T., and Pasalic, E. Meta-programming with built-in type equality. *Electron. Notes Theor. Comput. Sci. 199* (February 2008), 49–65.

[80] Stuart, T. *Understanding Computation: Impossible Code and the Meaning of Programs.* Understanding Computation. O'Reilly Media, Incorporated, 2013.

[81] Stump, A. Directly reflective meta-programming. *Higher Order Symbol. Comput. 22*, 2 (June 2009), 115–144.

[82] Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., and Donnelly, K. System F with type equality coercions. In *TLDI'07, ACM SIGPLAN Workshop on Types in Language Design and Implementation* (2007).

[83] Taha, W., Makholm, H., and Hughes, J. Tag elimination and Jones-optimality. In *Proceedings of PADO'01, Programs as Data Objects, Second Symposium* (2001), pp. 257–275.

[84] Taha, W., and Sheard, T. Metaml and multi-stage programming with explicit annotations. In *Theoretical Computer Science* (1999), ACM Press, pp. 203–217.

[85] Thiemann, P. Cogen in six lines. In *Proceedings of ICFP'96, ACM International Conference on Functional Programming* (1996), pp. 180–189.

[86] Trifonov, V., Saha, B., and Shao, Z. Fully reflexive intensional type analysis. *SIGPLAN Not. 35*, 9 (September 2000), 82–93.

[87] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *J. of Math 58*, 345-363 (1936), 5.

[88] Turner, D. Total functional programming. *Journal of Universal Computer Science 10* (2004).

[89] Vytiniotis, D., Washburn, G., and Weirich, S. An open and shut typecase. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New York, NY, USA, 2005), TLDI '05, ACM, pp. 13–24.

[90] Vytiniotis, D., and Weirich, S. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming 20*, 02 (2010), 175–210.

[91] Wadler, P. Theorems for free! In *Functional Programming Languages and Computer Architecture* (1989), ACM Press, pp. 347–359.

[92] Washburn, G., and Weirich, S. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2003), ICFP '03, ACM, pp. 249–262.

[93] Weirich, S. Type-safe cast: (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 58–67.

[94] Weirich, S. Higher-order intensional type analysis. In *Proceedings of the 11th European Symposium on Programming Languages and Systems* (London, UK, UK, 2002), ESOP '02, Springer-Verlag, pp. 98–114.

[95] Wikipedia. Rubinius. http://en.wikipedia.org/wiki/Rubinius, 2010.

[96] Wikipedia. Mogensen-Scott encoding. https://en.wikipedia.org/wiki/Mogensen–Scott_encoding, 2016.

[97] Wright, A., and Felleisen, M. A syntactic approach to type soundness. *Information and Computation 115*, 1 (1994), 38–94.

[98] Xi, H., Chen, C., and Chen, G. Guarded recursive datatype constructors. In *ACM SIGPLAN Notices* (2003), vol. 38, ACM, pp. 224–235.

[99] Yang, Z. Encoding types in ML-like languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 1998), ICFP '98, ACM, pp. 289–300.