Fully Reflexive Intensional Type Analysis*

Valery Trifonov   Bratin Saha   Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520-8285
{trifonov, saha, shao}@cs.yale.edu

ABSTRACT

Compilers for polymorphic languages can use runtime type inspection to support advanced implementation techniques such as tagless garbage collection, polymorphic marshalling, and flattened data structures. Intensional type analysis is a type-theoretic framework for expressing and certifying such type-analyzing computations. Unfortunately, existing approaches to intensional analysis do not work well on types with universal, existential, or fixpoint quantifiers. This makes it impossible to code applications such as garbage collection, persistence, or marshalling which must be able to examine the type of any runtime value.

We present a typed intermediate language that supports *fully reflexive* intensional type analysis. By fully reflexive, we mean that type-analyzing operations are applicable to the type of any runtime value in the language. In particular, we provide both type-level and term-level constructs for analyzing quantified types. Our system supports structural induction on quantified types yet type checking remains decidable. We show how to use reflexive type analysis to support type-safe marshalling and how to generate certified type-analyzing object code.

Keywords: certified code, runtime type dispatch, typed intermediate language.

1. INTRODUCTION

Runtime type analysis is used extensively in various applications and programming situations. Runtime services such as garbage collection and dynamic linking, applications such as marshalling and pickling, type-safe persistent programming, and unboxing implementations of polymorphic languages all analyze types to various degrees at runtime. Most existing compilers use untyped intermediate languages for compilation; therefore, they support runtime type inspection in a type-unsafe manner. In this paper, we present a statically typed intermediate language that allows runtime type analysis to be coded within the language. This allows us to leverage the power of dynamically typed languages, yet retain the advantages of static type checking.

Supporting runtime type analysis in a type-safe manner has been an active area of research. This paper builds on existing work [8] but makes the following new contributions:

- We support fully reflexive type analysis at the term level. Consequently, programs can analyze any runtime value such as function closures and polymorphic data structures.
- We support fully reflexive type analysis at the type level. Therefore, type transformations operating on arbitrary types can be encoded in our language.
- We prove that the language is sound and that type reduction is strongly normalizing and confluent.

In the companion technical report [18], we also show a translation into a language with type erasure semantics [2]. In a type preserving compiler this provides an approach to typed closure conversion which allows generation of certified object code.

2. MOTIVATION

The core issue that we address in this paper is the design of a statically typed intermediate language that supports runtime type analysis. Why is this important? Modern programming paradigms are increasingly giving rise to applications that rely critically on type information at runtime, for example:

- Java adopts dynamic linking as a key feature, and to ensure safe linking, an external module must be dynamically verified to satisfy the expected interface type.
- A garbage collector must keep track of all live heap objects, and for that type information must be kept at runtime to allow traversal of data structures.
- In a distributed computing environment, code and data on one machine may need to be pickled for transmission to a different machine, where the unpickler reconstructs the data structures from the bit stream. If the type of the data is not statically known at the destination (as is the case for the environment components of function closures), the unpickler must use type information, encoded in the bit stream, to correctly interpret the encoded value.
- Type-safe persistent programming requires language support for dynamic typing: the program must ensure that data read from a persistent store is of the expected type.

*This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," ARPA Order No. H559, issued under Contract No. F30602-99-1-0519, and in part by NSF Grants CCR-9633390 and CCR-9901011. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

ICFP'00, Montreal, Canada

- Finally, in polymorphic languages like ML, the type of a value may not be known statically; therefore, compilers have traditionally used inefficient, uniformly boxed data representation. To avoid this, several modern compilers [23, 19, 25] use runtime type information to support unboxed data representation.

When compiling code which uses runtime type inspections, most existing compilers use untyped intermediate languages, and reify runtime types into values at some early stage. However, discarding type information during compilation puts this approach at a serious disadvantage when it comes to generating certified code [13].

Code certification is appealing for a number of reasons. One need not trust the correctness of a compiler generating certified code; instead, one can verify the correctness of the generated code. Checking the correctness of a compiler-generated proof (of a program property) is much easier than proving the correctness of the compiler. Secondly, with the growth of web-based computing, programs are increasingly being developed at remote sites and shipped to clients for execution. Client programs may also download modules dynamically as they need them. For such a system to be practical, a client should be able to accept code from untrusted sources, but have a means of verifying it before execution. This again requires compilers that generate certified code.

A necessary step in building a certifying compiler is to have the compiler generate code that can be type-checked before execution. The type system ensures that the code accesses only the provided resources, makes legal function calls, *etc*. A certifying compiler can support runtime type analysis only in a typed framework.

The safety of such a system depends not only on the downloaded code, but also on the correctness of all the code that is executed by the system after type checking. This typically includes the runtime services like garbage collection, linking, *etc*. This code constitutes the trusted computing base of the system. Reducing the trusted computing base makes the system more reliable; for this, we must independently verify the correctness of this code. This implies that as many of the runtime services as possible should be written in a type-safe language, which requires support for runtime type analysis in a typed framework.

Finally, why is it important to have fully reflexive type analysis? Why do we want to analyze quantified types? Many type-analyzing applications mentioned above must handle arbitrary runtime values. For example, a pickler must be able to pickle any value, including closures (which have existential types), polymorphic functions, or recursive data structures. A garbage collector has to be able to traverse all data structures in the heap to track live objects. Therefore the language must support type analysis over any runtime value in the language.

2.1 Background

Harper and Morrisett [8] proposed intensional type analysis and presented a type-theoretic framework for expressing computations that analyze types at runtime. They introduced two explicit type-analysis operators: one at the term level (typecase) and another at the type level (Typerec); both use induction over the structure of types. Type-dependent primitive functions use these operators to analyze types and select the appropriate code. For example, a polymorphic subscript function for arrays might be written as the following pseudo-code:

$$
\begin{array}{l}
\mathsf{sub} = \Lambda\alpha.\,\mathsf{typecase}\ \alpha\ \mathsf{of} \\
\qquad\qquad \mathsf{int} \ \Rightarrow \mathsf{intsub} \\
\qquad\qquad \mathsf{real} \Rightarrow \mathsf{realsub} \\
\qquad\qquad \beta \quad\ \Rightarrow \mathsf{boxedsub}\,[\beta]
\end{array}
$$

$$
\begin{array}{lll}
(kinds) & \kappa ::= \Omega \mid \kappa \to \kappa' \\
(cons) & \tau ::= \mathsf{int} \mid \tau \to \tau' \mid \alpha \mid \lambda\alpha\!:\!\kappa.\,\tau \mid \tau\,\tau' \\
 & \quad\ \mid\ \mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to}) \\
(types) & \sigma ::= \tau \mid \forall\alpha\!:\!\kappa.\,\sigma
\end{array}
$$

Figure 1: The type language of Harper and Morrisett

Here sub analyzes the type α of the array elements and returns the appropriate subscript function. We assume that arrays of type int and real have specialized representations (defined by types, say, intarray and realarray), and therefore special subscript functions, while all other arrays use the default boxed representation.

Typing this subscript function is more interesting, because it must have all of the types $\mathsf{intarray} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{realarray} \to \mathsf{int} \to \mathsf{real}$, and $\forall\alpha.\,\mathsf{boxedarray}\,(\alpha) \to \mathsf{int} \to \alpha$. To assign a type to the subscript function, we need a construct at the type level that parallels the typecase analysis at the term level. In general, this facility is crucial since many type-analyzing operations like flattening and marshalling transform types in a non-uniform way. The subscript operation would then be typed as

$$
\begin{array}{rrcl}
 & \mathsf{sub} & : & \forall\alpha.\,\mathsf{Array}\,(\alpha) \to \mathsf{int} \to \alpha \\
\text{where} & \mathsf{Array} & = & \lambda\alpha.\,\mathsf{Typecase}\ \alpha\ \mathsf{of} \\
 & & & \qquad \mathsf{int} \ \Rightarrow \mathsf{intarray} \\
 & & & \qquad \mathsf{real} \Rightarrow \mathsf{realarray} \\
 & & & \qquad \beta \quad\ \Rightarrow \mathsf{boxedarray}\,\beta
\end{array}
$$

The Typecase construct in the above example is a special case of the Typerec construct in [8], which also supports primitive recursion over types.

2.2 The problem

The language of Harper and Morrisett only allows the analysis of monotypes; it does not support analysis of types with binding structure (*e.g.,* polymorphic, existential or recursive types). Therefore, type analyzing primitives that handle polymorphic code blocks, closures (since closures are represented as existentials [11]), or recursive structures, cannot be written in their language. The types in their language (in essence shown in Figure 1) are separated into two universes, *constructors* and *types*. The constructor calculus is a simply typed lambda calculus, with no polymorphic types. The Typerec operator analyzes only constructors of base kind Ω:

$$
\begin{array}{rcl}
\mathsf{int} & : & \Omega \\
\to & : & \Omega \to \Omega \to \Omega
\end{array}
$$

The kinds of these constructors' arguments do not contain any negative occurrence of the kind Ω, so int and $\to$ can be used to define Ω inductively. The Typerec operator is essentially an iterator over this inductive definition; its reduction rules can be written as:

$$
\begin{array}{l}
\mathsf{Typerec}\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to}) \leadsto \tau_{\mathsf{int}} \\
\mathsf{Typerec}\ (\tau_1 \to \tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to}) \leadsto \\
\quad \tau_{\to}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to}))\ (\mathsf{Typerec}\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to}))
\end{array}
$$

Here the Typerec operator examines the head constructor of the type being analyzed and chooses a branch accordingly. If the type is int, it reduces to the τ_{int} branch. If the type is $\tau_1 \to \tau_2$, the analysis proceeds recursively on the subtypes τ_1 and τ_2. The Typerec operator then applies the $\tau_{\to}$ branch to the original component types, and to the result of analyzing the components; thus providing a form of primitive recursion.

Types with binding structure can be constructed using higher-order abstract syntax. For example, the polymorphic type constructor $\forall$ can be given the kind $(\Omega \rightarrow \Omega) \rightarrow \Omega$, so that the type $\forall\alpha\!:\!\Omega.\,\alpha \rightarrow \alpha$ is represented as $\forall\,(\lambda\alpha\!:\!\Omega.\,\alpha \rightarrow \alpha)$. It would seem plausible to define an iterator with the reduction rule:

$$\begin{aligned}&\mathsf{Typerec}\ (\forall\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall})\\ &\leadsto\ \tau_{\forall}\,\tau\,(\lambda\alpha\!:\!\Omega.\ \mathsf{Typerec}\ \tau\,\alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}))\end{aligned}$$

However the negative occurrence of Ω in the kind of the argument of $\forall$ poses a problem: this iterator may fail to terminate! Consider the following example, assuming $\tau = \lambda\alpha\!:\!\Omega.\,\alpha$ and

$$\tau_{\forall} = \lambda\beta_1\!:\!\Omega \rightarrow \Omega.\,\lambda\beta_2\!:\!\Omega \rightarrow \Omega.\,\beta_2\,(\forall\beta_1)$$

the following reduction sequence will go on indefinitely:

$$\begin{aligned}&\mathsf{Typerec}\ (\forall\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall})\\ &\leadsto\ \tau_{\forall}\,\tau\,(\lambda\alpha\!:\!\Omega.\ \mathsf{Typerec}\ \tau\,\alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}))\\ &\leadsto\ \mathsf{Typerec}\ (\tau\,(\forall\,\tau))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall})\\ &\leadsto\ \mathsf{Typerec}\ (\forall\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall})\\ &\leadsto\ \ldots\end{aligned}$$

Clearly this makes typechecking Typerec undecidable.

Another serious problem in analyzing quantified types involves both the type-level and the term-level operators. Typed intermediate languages like FLINT [20] and TIL [24] are based on the calculus F_ω [5, 17], which has higher order type constructors. In a quantified type, say $\exists\alpha\!:\!\kappa.\,\tau$, the quantified variable α is no longer restricted to a base kind Ω, but can have an arbitrary kind κ. Consider the term-level typecase in such a scenario:

$$\begin{aligned}\mathsf{sub} = \Lambda\alpha.\ &\mathsf{typecase}\ \alpha\ \mathsf{of}\\ &\quad \mathsf{int} \qquad\ \Rightarrow e_{\mathsf{int}}\\ &\quad \ldots\\ &\quad \exists\alpha\!:\!\kappa.\,\tau \Rightarrow e_{\exists}\end{aligned}$$

To do anything useful in the $e_\exists$ branch, even to open a package of this type, we need to know the kind κ. We can get around this by having an infinite number of branches in the typecase, one for each kind; or by restricting type analysis to a finite set of kinds. Both of these approaches are clearly impractical. Recent work on typed compilation of ML and Java has shown that both would require an F_ω-like calculus with arbitrarily complex kinds [21, 22, 9].

2.3 Requirements for a solution

Before we discuss our solution, let us look at the properties we want it to have.

First, our language must support type analysis in the manner of Harper/Morrisett. That is, we want to include type analysis primitives that will analyze the entire syntax tree representing a type. Second, we want the analysis to continue inside the body of a quantified type; handling quantified types parametrically, or in a uniform way by providing a default case, is insufficient. As we will see later, many interesting type-directed operations require these two properties. Third, we do not want to restrict the kind of the (quantified) type variable in a quantified type; we want to analyze types where the quantification is over a variable of arbitrary kind.

Consider a type-directed pickler that converts a value of arbitrary type into an external representation. Suppose we want to pickle a closure. With a type-preserving compiler, the type of a closure would be represented as an existential with the environment held abstract. Even if the code is handled uniformly, the function must inspect the type of the environment (which is also the witness type of the existential package) to pickle it. This shows that at the term level, the analysis must proceed inside a quantified type. In Section 3.2, we show the encoding of a polymorphic equality function in our calculus; the comparison of existential values requires a similar technique.

The reason for not restricting the quantified type variable to a finite set of kinds is twofold. Restricting type analysis to a finite number of kinds would be *ad hoc* and there is no way of satisfactorily predetermining this finite set (this is even more the case when we compile Java into a typed intermediate language [9]). More importantly, if the kind of the bound variable is a known constant in the corresponding branch of the Typerec construct, it is easy to generalize the non-termination example of the previous section and break the decidability of the type system.

2.4 Our solution

The key problem in analyzing quantified types such as the polymorphic type $\forall\alpha : \Omega.\,\alpha \rightarrow \alpha$ is to determine what happens when the iteration reaches the quantified type variable α, or (in the general case of type variables of higher kinds) a normal form which is an application with a type variable in the head.

One approach would be to leave the type variable untouched while analyzing the body of the quantified type. The equational theory of the type language then includes a reduction of the form $(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots) \leadsto \alpha$ so that the iterator vanishes when it reaches a type variable. However this would break the confluence of the type language—the application of $\lambda\alpha : \Omega.\ \mathsf{Typerec}\ \alpha\ \mathsf{of} \ldots$ to τ would reduce in general to different types if we perform the β-reduction step first or eliminate the iterator first.

Crary and Weirich [1] propose another method for solving this problem. Their language LX allows the representation of terms with bound variables using deBruijn notation and an encoding of natural numbers as types. To analyze quantified types, the iterator carries an environment mapping indices to types; when the iterator reaches a type variable, it returns the corresponding type from the environment. This method has several disadvantages.

- It is not fully reflexive, since it does not allow analysis of all quantified types—their analysis is restricted to types with quantification only over variables of kind Ω.
- The technique is “limited to *parametrically* polymorphic functions, and cannot account for functions that perform intensional type analysis” [1, Section 4.1]. For example polymorphic types such as $\forall\alpha\!:\!\Omega.\ \mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots$ are not analyzable in their framework.
- The correctness of the structure of a type encoded using deBruijn notation cannot be verified by the kind language (indices not corresponding to bound variables go undetected, so the environment must provide a default type for them), which does not break the type soundness but opens the door for programmer mistakes.

To account for non-parametrically polymorphic functions, we must analyze the quantified type variable. Moreover, we want to have confluence of the type language, so β-reduction should be transparent to the iterator. This is possible only if the analysis gets suspended when it reaches a type variable, or its application, of kind Ω, and resumes when the variable gets substituted. Therefore, we consider $(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots)$ to be a normal form. For example, the result of analyzing the body $(\alpha \rightarrow \mathsf{int})$ of the polymorphic type $\forall\alpha\!:\!\kappa.\,\alpha \rightarrow \mathsf{int}$ is

$$\begin{aligned}&\mathsf{Typerec}\ (\alpha \rightarrow \mathsf{int})\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}) \leadsto\\ &\tau_{\rightarrow}\,\alpha\,\mathsf{int}\,(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}))\,(\tau_{\mathsf{int}})\end{aligned}$$

We formalize the analysis of quantified types when we present the type reduction rules of the $\mathsf{Typerec}$ construct (Figure 5).

The other problem is to analyze quantified types when the quantified variable can be of an arbitrary kind. In our language the solution is similar at both the type and the term levels: we use kind polymorphism! We introduce kind abstractions at the type level ($\Lambda\chi.\,\tau$) and at the term level ($\Lambda^{+}\chi.\,e$) to bind the kind of the quantified variable. (See Section 3 for details.)

Kind polymorphism also ensures the termination of the $\mathsf{Typerec}$ constructor. Consider again the analysis of the polymorphic type:

$$\begin{array}{l}\mathsf{Typerec}\ (\forall\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall})\\ \rightsquigarrow\ \tau_{\forall}\,\tau\,(\lambda\alpha\!:\!\Omega.\ \mathsf{Typerec}\ \tau\,\alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}))\end{array}$$

Informally, we must ensure that the type being analyzed decreases in size at every iteration. That is $\tau\alpha$ is smaller than $\forall\tau$. (Note that the previous non-terminating example violates this requirement). This will be true if we can ensure that α is always substituted by a single variable. Therefore, we make the kind of α abstract by using kind polymorphism; α now has the kind bound in the $\tau_{\forall}$ branch. The only way to construct another type of this kind is to bind a type variable of the same kind in the $\tau_{\forall}$ branch. This ensures that α can only be substituted by another type variable.

It is important to note that our language provides no facilities for kind analysis. Analyzing the kind κ of the bound variable α in the type $\forall\,(\lambda\alpha:\kappa.\,\tau)$ would let us synthesize a type argument of the same kind, for every kind κ. The synthesized type can then be used in the style of the non-termination example of the previous section. Intuitively, we would not be able to guarantee that the type being analyzed decreases at every step.

The rest of the paper is organized as follows. Section 3 describes the language λ_i^P supporting analysis of polymorphic and existential types. Section 4 presents the language λ_i^Q that also includes support for analysis of recursive types. In the companion technical report [18] we also show a translation into a language with type erasure semantics [2].

3. ANALYZING POLYMORPHIC TYPES

In the impredicative F_ω calculus, the polymorphic types $\forall\alpha\!:\!\kappa.\,\tau$ can be viewed as generated by an infinite set of type constructors $\forall_\kappa$ of kind $(\kappa\rightarrow\Omega)\rightarrow\Omega$, one for each kind κ. The type $\forall\alpha\!:\!\kappa.\,\tau$ is then represented as $\forall_\kappa\,(\lambda\alpha\!:\!\kappa.\,\tau)$. The kinds of constructors that can generate types of kind Ω then would be

$$\begin{array}{lcl}\mathsf{int} & : & \Omega\\ \twoheadrightarrow & : & \Omega\rightarrow\Omega\rightarrow\Omega\\ \forall_\Omega & : & (\Omega\rightarrow\Omega)\rightarrow\Omega\\ \ldots & & \\ \forall_\kappa & : & (\kappa\rightarrow\Omega)\rightarrow\Omega\\ \ldots & & \end{array}$$

We can avoid the infinite number of $\forall_\kappa$ constructors by defining a single constructor $\forall$ of polymorphic kind $\forall\chi.\,(\chi\rightarrow\Omega)\rightarrow\Omega$ and then instantiating it to a specific kind before forming polymorphic types. More importantly, this technique also removes the negative occurrence of Ω from the kind of the argument of the constructor $\forall_\Omega$. Hence in our λ_i^P calculus we extend F_ω with polymorphic kinds and add a type constant $\forall$ of kind $\forall\chi.\,(\chi\rightarrow\Omega)\rightarrow\Omega$ to the type language. The polymorphic type $\forall\alpha\!:\!\kappa.\,\tau$ is now represented as $\forall\,[\kappa]\,(\lambda\alpha\!:\!\kappa.\,\tau)$.

We define the syntax of the λ_i^P calculus in Figure 2, and some derived forms of types in Figure 3. The static semantics of λ_i^P is shown in Figures 4 and 5 as a set of rules for judgments using the following environments:

$$\begin{array}{llcl}\text{kind environment} & \mathcal{E} & ::= & \varepsilon\ \mid\ \mathcal{E},\chi\\ \text{type environment} & \Delta & ::= & \varepsilon\ \mid\ \Delta,\alpha\!:\!\kappa\\ \text{term environment} & \Gamma & ::= & \varepsilon\ \mid\ \Gamma,x\!:\!\tau\end{array}$$

The $\mathsf{Typerec}$ operator analyzes polymorphic types with bound variables of arbitrary kind. The corresponding branch of the operator must bind the kind of the quantified type variable; for that purpose the language provides kind abstraction ($\Lambda\chi.\,\tau$) and kind application ($\tau\,[\kappa]$) at the type level. The formation rules for these constructs, excerpted from Figure 4, are

$$\frac{\mathcal{E},\chi;\Delta\vdash\tau\ :\ \kappa}{\mathcal{E};\Delta\vdash\Lambda\chi.\,\tau\ :\ \forall\chi.\,\kappa}\qquad\frac{\mathcal{E};\Delta\vdash\tau\ :\ \forall\chi.\,\kappa\qquad\mathcal{E}\vdash\kappa'}{\mathcal{E};\Delta\vdash\tau\,[\kappa']\ :\ \kappa\{\kappa'/\chi\}}$$

Similarly, while analyzing a polymorphic type, the term-level construct $\mathsf{typecase}$ must bind the kind of the quantified type variable. Therefore, we introduce kind abstraction ($\Lambda^{+}\chi.\,e$) and kind application ($e\,[\kappa]^{+}$) at the term level. To type the term-level kind abstraction, we need a type construct $\forall^{+}\chi.\,\tau$ that binds the kind variable χ in the type τ. The formation rules are shown below.

$$\frac{\mathcal{E},\chi;\Delta;\Gamma\vdash v\ :\ \tau}{\mathcal{E};\Delta;\Gamma\vdash\Lambda^{+}\chi.\,v\ :\ \forall^{+}\chi.\,\tau}\qquad\frac{\mathcal{E};\Delta;\Gamma\vdash e\ :\ \forall^{+}\chi.\,\tau\quad\mathcal{E}\vdash\kappa}{\mathcal{E};\Delta;\Gamma\vdash e\,[\kappa]^{+}\ :\ \tau\{\kappa/\chi\}}$$

However, since our goal is fully reflexive type analysis, we need to analyze kind-polymorphic types as well. As with polymorphic types, we can represent the type $\forall^{+}\chi.\,\tau$ as the application of a type constructor $\forall^{+}$ of kind $(\forall\chi.\,\Omega)\rightarrow\Omega$ to a kind abstraction $\Lambda\chi.\,\tau$. Thus the kinds of the constructors for types of kind Ω are

$$\begin{array}{rcl}\mathsf{int} & : & \Omega\\ \twoheadrightarrow & : & \Omega\rightarrow\Omega\rightarrow\Omega\\ \forall & : & \forall\chi.\,(\chi\rightarrow\Omega)\rightarrow\Omega\\ \forall^{+} & : & (\forall\chi.\,\Omega)\rightarrow\Omega\end{array}$$

None of these constructors' arguments have the kind Ω in a negative position; hence the kind Ω can now be defined inductively in terms of these constructors. The $\mathsf{Typerec}$ construct is then the iterator over this kind. The formation rule for $\mathsf{Typerec}$ follows naturally

$$\begin{array}{llcl}(\mathit{kinds}) & \kappa & ::= & \Omega\ \mid\ \kappa\rightarrow\kappa'\ \mid\ \chi\ \mid\ \forall\chi.\,\kappa\\ \\ (\mathit{types}) & \tau & ::= & \mathsf{int}\ \mid\ \twoheadrightarrow\ \mid\ \forall\ \mid\ \forall^{+}\\ & & \mid & \alpha\ \mid\ \Lambda\chi.\,\tau\ \mid\ \lambda\alpha\!:\!\kappa.\,\tau\ \mid\ \tau\,[\kappa]\ \mid\ \tau\,\tau'\\ & & \mid & \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^{+}})\\ \\ (\mathit{values}) & v & ::= & i\ \mid\ \Lambda^{+}\chi.\,e\ \mid\ \Lambda\alpha\!:\!\kappa.\,e\ \mid\ \lambda x\!:\!\tau.\,e\ \mid\ \mathsf{fix}\,x\!:\!\tau.\,v\\ \\ (\mathit{terms}) & e & ::= & v\ \mid\ x\ \mid\ e\,[\kappa]^{+}\ \mid\ e\,[\tau]\ \mid\ e\,e'\\ & & \mid & \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}})\end{array}$$

Figure 2: Syntax of the λ_i^P language

$$\begin{array}{c}\tau\rightarrow\tau'\equiv((\twoheadrightarrow)\,\tau)\,\tau'\\ \forall\alpha\!:\!\kappa.\,\tau\equiv(\forall\,[\kappa])\,(\lambda\alpha\!:\!\kappa.\,\tau)\\ \forall^{+}\chi.\,\tau\equiv\forall^{+}(\Lambda\chi.\,\tau)\end{array}$$

Figure 3: Syntactic sugar for λ_i^P types

Kind formation $\quad \mathcal{E} \vdash \kappa$

$$\mathcal{E} \vdash \Omega \qquad \frac{\chi \in \mathcal{E}}{\mathcal{E} \vdash \chi} \qquad \frac{\mathcal{E} \vdash \kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E} \vdash \kappa \to \kappa'} \qquad \frac{\mathcal{E}, \chi \vdash \kappa}{\mathcal{E} \vdash \forall \chi.\, \kappa}$$

Type environment formation $\quad \mathcal{E} \vdash \Delta$

$$\mathcal{E} \vdash \varepsilon \qquad \frac{\mathcal{E} \vdash \Delta \quad \mathcal{E} \vdash \kappa}{\mathcal{E} \vdash \Delta, \alpha{:}\kappa}$$

Type formation $\quad \mathcal{E}; \Delta \vdash \tau \,:\, \kappa$

$$\frac{\mathcal{E} \vdash \Delta}{\begin{array}{ll} \mathcal{E}; \Delta \vdash \mathsf{int} & : \Omega \\ \mathcal{E}; \Delta \vdash (\twoheadrightarrow) & : \Omega \to \Omega \to \Omega \\ \mathcal{E}; \Delta \vdash \boldsymbol{\forall} & : \forall \chi.\, (\chi \to \Omega) \to \Omega \\ \mathcal{E}; \Delta \vdash \boldsymbol{\forall}^{+} & : (\forall \chi.\, \Omega) \to \Omega \end{array}} \qquad \frac{\mathcal{E} \vdash \Delta \quad \alpha{:}\kappa \text{ in } \Delta}{\mathcal{E}; \Delta \vdash \alpha \,:\, \kappa}$$

$$\frac{\mathcal{E}, \chi; \Delta \vdash \tau \,:\, \kappa}{\mathcal{E}; \Delta \vdash \Lambda \chi.\, \tau \,:\, \forall \chi.\, \kappa} \qquad \frac{\mathcal{E}; \Delta \vdash \tau \,:\, \forall \chi.\, \kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash \tau\,[\kappa'] \,:\, \kappa\{\kappa'/\chi\}}$$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa \vdash \tau \,:\, \kappa'}{\mathcal{E}; \Delta \vdash \lambda \alpha{:}\kappa.\, \tau \,:\, \kappa \to \kappa'} \qquad \frac{\mathcal{E}; \Delta \vdash \tau \,:\, \kappa' \to \kappa \quad \mathcal{E}; \Delta \vdash \tau' \,:\, \kappa'}{\mathcal{E}; \Delta \vdash \tau\, \tau' \,:\, \kappa}$$

$$\frac{\begin{array}{ll} \mathcal{E}; \Delta \vdash \tau & : \Omega \\ \mathcal{E}; \Delta \vdash \tau_{\mathsf{int}} & : \kappa \\ \mathcal{E}; \Delta \vdash \tau_{\to} & : \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \mathcal{E}; \Delta \vdash \tau_{\forall} & : \forall \chi.\, (\chi \to \Omega) \to (\chi \to \kappa) \to \kappa \\ \mathcal{E}; \Delta \vdash \tau_{\forall^{+}} & : (\forall \chi.\, \Omega) \to (\forall \chi.\, \kappa) \to \kappa \end{array}}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; \tau \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \,:\, \kappa}$$

Term environment formation $\quad \mathcal{E}; \Delta \vdash \Gamma$

$$\frac{\mathcal{E} \vdash \Delta}{\mathcal{E}; \Delta \vdash \varepsilon} \qquad \frac{\mathcal{E}; \Delta \vdash \Gamma \quad \mathcal{E}; \Delta \vdash \tau \,:\, \Omega}{\mathcal{E}; \Delta \vdash \Gamma, x{:}\tau}$$

Term formation $\quad \mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau \quad \mathcal{E}; \Delta \vdash \tau \rightsquigarrow \tau' \,:\, \Omega}{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau'} \qquad \frac{\mathcal{E}; \Delta \vdash \Gamma}{\mathcal{E}; \Delta; \Gamma \vdash i \,:\, \mathsf{int}}$$

$$\frac{\mathcal{E}; \Delta \vdash \Gamma \quad x{:}\tau \text{ in } \Gamma}{\mathcal{E}; \Delta; \Gamma \vdash x \,:\, \tau} \qquad \frac{\mathcal{E}, \chi; \Delta; \Gamma \vdash v \,:\, \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda^{+} \chi.\, v \,:\, \forall^{+} \chi.\, \tau}$$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash v \,:\, \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda \alpha{:}\kappa.\, v \,:\, \forall \alpha{:}\kappa.\, \tau} \qquad \frac{\mathcal{E}; \Delta; \Gamma, x{:}\tau \vdash e \,:\, \tau'}{\mathcal{E}; \Delta; \Gamma \vdash \lambda x{:}\tau.\, e \,:\, \tau \to \tau'}$$

$$\frac{\begin{array}{c} \mathcal{E}; \Delta; \Gamma, x{:}\tau \vdash v \,:\, \tau \\ \tau = \forall^{+} \chi_1 \ldots \chi_n.\, \forall \alpha_1{:}\kappa_1 \ldots \alpha_m{:}\kappa_m{:}\tau_1 \to \tau_2. \\ n \geq 0, m \geq 0 \end{array}}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{fix}\, x{:}\tau.\, v \,:\, \tau}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \boldsymbol{\forall}^{+} \tau \quad \mathcal{E} \vdash \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\,[\kappa]^{+} \,:\, \tau\,[\kappa]}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \boldsymbol{\forall}\,[\kappa]\, \tau \quad \mathcal{E}; \Delta \vdash \tau' \,:\, \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\,[\tau'] \,:\, \tau\, \tau'}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau' \to \tau \quad \mathcal{E}; \Delta; \Gamma \vdash e' \,:\, \tau'}{\mathcal{E}; \Delta; \Gamma \vdash e\, e' \,:\, \tau}$$

$$\frac{\begin{array}{ll} \mathcal{E}; \Delta \vdash \tau & : \Omega \to \Omega \\ \mathcal{E}; \Delta \vdash \tau' & : \Omega \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\mathsf{int}} & : \tau\, \mathsf{int} \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\to} & : \forall \alpha{:}\Omega.\, \forall \alpha'{:}\Omega.\, \tau\, (\alpha \to \alpha') \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\forall} & : \forall^{+} \chi.\, \forall \alpha{:}\chi \to \Omega.\, \tau\, (\boldsymbol{\forall}\,[\chi]\, \alpha) \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\forall^{+}} & : \forall \alpha{:}(\forall \chi.\, \Omega).\, \tau\, (\boldsymbol{\forall}^{+} \alpha) \end{array}}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{typecase}[\tau]\; \tau' \;\mathsf{of}\; (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^{+}}) \,:\, \tau\, \tau'}$$

Figure 4: Formation rules of λ_i^P

Type reduction $\quad \mathcal{E}; \Delta \vdash \tau_1 \rightsquigarrow \tau_2 \,:\, \kappa$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa' \vdash \tau \,:\, \kappa \quad \mathcal{E}; \Delta \vdash \tau' \,:\, \kappa'}{\mathcal{E}; \Delta \vdash (\lambda \alpha{:}\kappa'.\, \tau)\, \tau' \rightsquigarrow \tau\{\tau'/\alpha\} \,:\, \kappa}$$

$$\frac{\mathcal{E}, \chi; \Delta \vdash \tau \,:\, \forall \chi.\, \kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash (\Lambda \chi.\, \tau)\,[\kappa'] \rightsquigarrow \tau\{\kappa'/\chi\} \,:\, \kappa\{\kappa'/\chi\}}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau \,:\, \kappa \to \kappa' \quad \alpha \notin \mathit{ftv}(\tau)}{\mathcal{E}; \Delta \vdash \lambda \alpha{:}\kappa.\, \tau\, \alpha \rightsquigarrow \tau \,:\, \kappa \to \kappa'}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau \,:\, \forall \chi'.\, \kappa \quad \chi \notin \mathit{fkv}(\tau)}{\mathcal{E}; \Delta \vdash \Lambda \chi.\, \tau\,[\chi] \rightsquigarrow \tau \,:\, \forall \chi'.\, \kappa}$$

$$\frac{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; \mathsf{int} \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \,:\, \kappa}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; \mathsf{int} \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau_{\mathsf{int}} \,:\, \kappa}$$

$$\frac{\begin{array}{c} \mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; \tau_1 \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau_1' \,:\, \kappa \\ \mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; \tau_2 \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau_2' \,:\, \kappa \end{array}}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; ((\twoheadrightarrow)\, \tau_1\, \tau_2) \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau_{\to}\, \tau_1\, \tau_2\, \tau_1'\, \tau_2' \,:\, \kappa}$$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa' \vdash \mathsf{Typerec}[\kappa]\; (\tau\, \alpha) \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau' \,:\, \kappa}{\begin{array}{c} \mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; (\boldsymbol{\forall}\,[\kappa']\, \tau) \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \\ \rightsquigarrow \tau_{\forall}\,[\kappa']\, \tau\, (\lambda \alpha{:}\kappa'.\, \tau') \,:\, \kappa \end{array}}$$

$$\frac{\mathcal{E}, \chi; \Delta \vdash \mathsf{Typerec}[\kappa]\; (\tau\,[\chi]) \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau' \,:\, \kappa}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\; (\boldsymbol{\forall}^{+} \tau) \;\mathsf{of}\; (\tau_{\mathsf{int}};\, \tau_{\to};\, \tau_{\forall};\, \tau_{\forall^{+}}) \rightsquigarrow \tau_{\forall^{+}}\, \tau\, (\Lambda \chi.\, \tau') \,:\, \kappa}$$

Figure 5: Selected λ_i^P type reduction rules

from the type reduction rules (Figure 5). Depending on the head constructor of the type being analyzed, Typerec chooses one of the branches. At the int type, it returns the τ_{int} branch. At the function type $\tau \to \tau'$, it applies the $\tau_{\to}$ branch to the components τ and τ' and to the result of the iteration over τ and τ'.

When analyzing a polymorphic type, the reduction rule is

$$\begin{array}{l}\mathsf{Typerec}[\kappa]\ (\forall\alpha\!:\!\kappa'.\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \\ \tau_{\forall}\ [\kappa']\ (\lambda\alpha\!:\!\kappa'.\,\tau)\ (\lambda\alpha\!:\!\kappa'.\,\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{array}$$

Thus the $\forall$-branch of Typerec receives as arguments the kind of the bound variable, the abstraction representing the quantified type, and a type function encapsulating the result of the iteration on the body of the quantified type. Since $\tau_{\forall}$ must be parametric in the kind κ' (there are no facilities for kind analysis in the language), it can only apply its second and third arguments to locally introduced type variables of kind κ'. We believe this restriction, which is crucial for preserving strong normalization of the type language, is quite reasonable in practice. For instance $\tau_{\forall}$ can yield a quantified type based on the result of the iteration.

The reduction rule for analyzing a kind-polymorphic type is

$$\begin{array}{l}\mathsf{Typerec}[\kappa]\ (\forall^{+}\chi.\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \\ \tau_{\forall^+}\ (\Lambda\chi.\,\tau)\ (\Lambda\chi.\,\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{array}$$

The arguments of the $\tau_{\forall^+}$ are the kind abstraction underlying the kind-polymorphic type and a kind abstraction encapsulating the result of the iteration on the body of the quantified type.

For ease of presentation, we will use ML-style pattern matching syntax to define a type involving Typerec. Instead of

$$\begin{array}{rl}\tau = & \lambda\alpha\!:\!\Omega.\,\mathsf{Typerec}[\kappa]\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \\ \text{where} & \tau_{\to} = \lambda\alpha_1\!:\!\Omega.\,\lambda\alpha_2\!:\!\Omega.\,\lambda\alpha'_1\!:\!\kappa.\,\lambda\alpha'_2\!:\!\kappa.\,\tau'_{\to} \\ & \tau_{\forall} = \Lambda\chi.\,\lambda\alpha\!:\!\chi \to \Omega.\,\lambda\alpha'\!:\!\chi \to \kappa.\,\tau'_{\forall} \\ & \tau_{\forall^+} = \lambda\alpha\!:\!(\forall\chi.\,\Omega).\,\lambda\alpha'\!:\!(\forall\chi.\,\kappa).\,\tau'_{\forall^+}\end{array}$$

we will write

$$\begin{array}{lcl}\tau\ (\mathsf{int}) & = & \tau_{\mathsf{int}} \\ \tau\ (\alpha_1 \to \alpha_2) & = & \tau'_{\to}\{\tau\ (\alpha_1), \tau\ (\alpha_2)/\alpha'_1, \alpha'_2\} \\ \tau\ (\forall\,[\chi]\,\alpha_1) & = & \tau'_{\forall}\{\lambda\alpha\!:\!\chi.\,\tau\ (\alpha_1\,\alpha)/\alpha'\} \\ \tau\ (\forall^{+}\alpha_1) & = & \tau'_{\forall^+}\{\Lambda\chi.\,\tau\ (\alpha_1\,[\chi])/\alpha'\}\end{array}$$

To illustrate the type-level analysis we will use the Typerec operator to define the class of types admitting equality comparisons. To make the example non-trivial we extend the language with a product type constructor $\times$ of the same kind as $\to$, and with existential types with type constructor $\exists$ of kind identical to that of $\forall$, writing $\exists\alpha\!:\!\kappa.\,\tau$ for $\exists\,[\kappa]\,(\lambda\alpha\!:\!\kappa.\,\tau)$. Correspondingly we extend Typerec with a product branch $\tau_{\times}$ and an existential branch $\tau_{\exists}$ which behave in exactly the same way as the $\tau_{\to}$ branch and the $\tau_{\forall}$ branch respectively. We will use Bool instead of int.

A polymorphic function eq comparing two objects for equality is not defined on values of function or polymorphic types. We can enforce this restriction statically if we define a type operator Eq of kind $\Omega \to \Omega$, which maps function and polymorphic types to the type $\mathsf{Void} \equiv \forall\alpha : \Omega.\,\alpha$ (a type with no values), and require the arguments of eq to be of type $\mathsf{Eq}\,\tau$ for some type τ. Thus, given any type τ, the function Eq serves to verify that a non-equality type does not occur inside τ.

$$\begin{array}{lcl}\mathsf{Eq}\ (\mathsf{Bool}) & = & \mathsf{Bool} \\ \mathsf{Eq}\ (\alpha_1 \to \alpha_2) & = & \mathsf{Void} \\ \mathsf{Eq}\ (\alpha_1 \times \alpha_2) & = & \mathsf{Eq}\ (\alpha_1) \times \mathsf{Eq}\ (\alpha_2) \\ \mathsf{Eq}\ (\forall\,[\chi]\,\alpha) & = & \mathsf{Void} \\ \mathsf{Eq}\ (\forall^{+}\alpha) & = & \mathsf{Void} \\ \mathsf{Eq}\ (\exists\,[\chi]\,\alpha) & = & \exists\,[\chi]\,(\lambda\alpha_1\!:\!\chi.\,\mathsf{Eq}\ (\alpha\,\alpha_1))\end{array}$$

$$(\lambda x\!:\!\tau.\,e)\ v \rightsquigarrow e\{v/x\} \qquad (\mathsf{fix}\,x\!:\!\tau.\,v)\ v' \rightsquigarrow (v\{\mathsf{fix}\,x\!:\!\tau.\,v/x\})\ v'$$

$$(\Lambda\alpha\!:\!\kappa.\,v)[\tau] \rightsquigarrow v\{\tau/\alpha\} \qquad (\mathsf{fix}\,x\!:\!\tau.\,v)[\tau] \rightsquigarrow (v\{\mathsf{fix}\,x\!:\!\tau.\,v/x\})[\tau]$$

$$(\Lambda^{+}\chi.\,v)[\kappa]^{+} \rightsquigarrow v\{\kappa/\chi\} \quad (\mathsf{fix}\,x\!:\!\tau.\,v)[\kappa]^{+} \rightsquigarrow (v\{\mathsf{fix}\,x\!:\!\tau.\,v/x\})[\kappa]^{+}$$

$$\frac{e \rightsquigarrow e'}{e\,e_1 \rightsquigarrow e'\,e_1} \qquad \frac{e \rightsquigarrow e'}{v\,e \rightsquigarrow v\,e'} \qquad \frac{e \rightsquigarrow e'}{e[\tau] \rightsquigarrow e'[\tau]} \qquad \frac{e \rightsquigarrow e'}{e[\kappa]^{+} \rightsquigarrow e'[\kappa]^{+}}$$

$$\mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\mathsf{int}}$$

$$\mathsf{typecase}[\tau]\ (\tau_1 \to \tau_2)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\to}\ [\tau_1]\ [\tau_2]$$

$$\mathsf{typecase}[\tau]\ (\forall\,[\kappa]\ \tau)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\forall}\ [\kappa]^{+}\ [\tau]$$

$$\mathsf{typecase}[\tau]\ (\forall^{+}\tau)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\forall^+}\ [\tau]$$

$$\frac{\varepsilon;\varepsilon \vdash \tau' \rightsquigarrow^{*} \nu'\!:\!\Omega \qquad \nu' \text{ is a normal form}}{\begin{array}{l}\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow \\ \quad \mathsf{typecase}[\tau]\ \nu'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+})\end{array}}$$

Figure 6: Operational semantics of λ_i^P

The property is enforced even on hidden types in an existentially typed package by the reduction rule for Typerec which suspends its action on normal forms with variable head. For instance a term e can only be given type

$$\mathsf{Eq}\ (\exists\alpha\!:\!\Omega.\,\alpha \times \alpha) = \exists\alpha\!:\!\Omega.\,\mathsf{Eq}\,\alpha \times \mathsf{Eq}\,\alpha$$

if it can be shown that e is a pair of terms of type $\mathsf{Eq}\,\tau$ for some τ, *i.e.,* terms of equality type. Note that $\mathsf{Eq}\ ((\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Bool} \to \mathsf{Bool}))$ reduces to $(\mathsf{Void} \times \mathsf{Void})$; a more complicated definition is necessary to map this type to Void.

At the term level type analysis is carried out by the typecase construct; however, it is not iterative since the term language has a recursion primitive, fix. The $e_{\forall}$ branch of typecase binds the kind and the type abstraction carried by the type constructor $\forall$, while the $e_{\forall^+}$ branch binds the kind abstraction carried by $\forall^{+}$.

$$\begin{array}{l}\mathsf{typecase}[\tau]\ (\forall\,[\kappa]\ \tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\forall}\ [\kappa]^{+}\ [\tau'] \\ \mathsf{typecase}[\tau]\ (\forall^{+}\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\forall^+}\ [\tau']\end{array}$$

The operational semantics of the term language of λ_i^P is presented in Figure 6.

The language λ_i^P has the following important properties (for detailed proofs we refer the reader to the companion technical report [18]).

THEOREM 3.1. *Reduction of well-formed types is strongly normalizing.*

We prove strong normalization of the type language following Girard's method of candidates [6], using his definition of a candidate. The standard set of neutral types is extended to include types constructed by Typerec. We define R_{Ω} as the set of types τ of kind Ω such that the type $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to a candidate for kind κ whenever the branches belong to candidates of the corresponding kinds from the Typerec formation rule. We then prove that this set is a candidate. Next we define the set $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ of types of kind κ (for given candidates $\overline{\mathcal{C}}$ corresponding to the free kind variables $\overline{\chi}$ of κ), equal to R_{Ω} for kind

Ω, and defined inductively as in [6] for function, polymorphic, and variable kinds. We show that $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ is a candidate. Finally we prove that $\mathcal{S}_\bullet[\overline{\mathcal{C}}/\overline{\chi}]$ is closed under substitution of types for free type variables; strong normalization is an immediate corollary.

THEOREM 3.2. *Reduction of well-formed types is confluent.*

Confluence of type reduction is a corollary of local confluence, which we prove by case analysis of the type reduction relation ($\leadsto$). We consider type contexts with two holes and show that the reduction is locally confluent in each case.

We say that a term e is stuck if e is not a value and $e \leadsto e'$ for no term e'.

THEOREM 3.3 (SOUNDNESS OF λ_i^P FOR TYPE SAFETY). *If $\varepsilon;\ \varepsilon;\ \varepsilon \vdash e : \tau$ and $e \leadsto^* e'$ in λ_i^P, then e' is not stuck.*

We prove soundness of the system using a contextual semantics in Wright/Felleisen style [26] using the standard progress, subject reduction, and substitution lemmas as well as the confluence and strong normalization properties of the λ_i^P type system.

3.1 Example: Marshalling

One of the examples that Harper and Morrisett [8] use to illustrate the power of intensional type analysis is based on the extension of ML for distributed computing proposed by Ohori and Kato [14]. The idea is to convert values into a form which can be used for transmission over a network. An integer value may be transmitted directly, but a function may not; instead, a globally unique identifier is transmitted that serves as a proxy at the remote site. These identifiers are associated with their functions by a name server that may be contacted through a primitive addressing scheme. The remote sites use the identifiers to make remote calls to the function. Harper and Morrisett show how to define types of transmissible values as well as functions for marshalling to and unmarshalling from these types using intensional type analysis. However, the predicativity of their calculus prevents them from handling the full calculus of Ohori and Kato, which also includes the remote representation of polymorphic functions and remote type application.

In λ_i^P marshalling of polymorphic values is straightforward; in fact it offers more flexibility than the calculus of Ohori and Kato needs, since polymorphic functions become first-class values, and polymorphic types can be used in remote type applications. Adapting the constructs of [8] to λ_i^P, we introduce a type constructor $\mathsf{Id} : \Omega \to \Omega$. A value of type τ has a global identifier of type $\mathsf{Id}\,\tau$. The $\mathsf{Typerec}$ and $\mathsf{typecase}$ operators are extended in an obvious way. For example, the following type reduction relation is added:

$$\begin{array}{l}\mathsf{Typerec}[\kappa]\ (\mathsf{Id}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{Id}) \leadsto \\ \quad \tau_{Id}\,\tau\ (\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{Id}))\end{array}$$

The type of the remote representation of values of type τ is $\mathsf{Tran}\,\tau$, defined in [8] using intensional analysis of τ. Values of type $\mathsf{Tran}\,\tau$ do not contain any abstractions; all the abstractions are wrapped inside an Id constructor. We can extend the Harper/Morrisett definition of Tran to handle the quantified types of λ_i^P as follows:

$$\begin{array}{lcl}
\mathsf{Tran}\,(\mathsf{int}) & = & \mathsf{int} \\
\mathsf{Tran}\,(\alpha_1 \to \alpha_2) & = & \mathsf{Id}\,(\mathsf{Tran}\,\alpha_1 \to \mathsf{Tran}\,\alpha_2) \\
\mathsf{Tran}\,(\forall\,[\chi]\,\alpha) & = & \mathsf{Id}\,(\forall\alpha' : \chi.\,(\lambda\alpha_1 : \chi.\,\mathsf{Tran}\,(\alpha\,\alpha_1))\,\alpha') \\
\mathsf{Tran}\,(\forall^+\alpha) & = & \mathsf{Id}\,(\forall^+\chi'.\,(\Lambda\chi.\,\mathsf{Tran}\,(\alpha\,[\chi]))\,[\chi']) \\
\mathsf{Tran}\,(\mathsf{Id}\,\alpha) & = & \mathsf{Id}\,\alpha
\end{array}$$

At the term level the system provides primitives for creating global identifiers and performing remote invocations:[1]

$$\begin{array}{r}
\mathsf{newid} : \forall\alpha_1 : \Omega.\,\forall\alpha_2 : \Omega.\,(\mathsf{Tran}\,\alpha_1 \to \mathsf{Tran}\,\alpha_2) \to \mathsf{Tran}\,(\alpha_1 \to \alpha_2) \\
\mathsf{rapp} : \forall\alpha_1 : \Omega.\,\forall\alpha_2 : \Omega.\,\mathsf{Tran}\,(\alpha_1 \to \alpha_2) \to \mathsf{Tran}\,\alpha_1 \to \mathsf{Tran}\,\alpha_2 \\
\mathsf{newpid} : \forall^+\chi.\,\forall\alpha : \chi \to \Omega.\,(\forall\alpha' : \chi.\,\mathsf{Tran}\,(\alpha\,\alpha')) \to \mathsf{Tran}\,(\forall\,[\chi]\,\alpha) \\
\mathsf{rtapp} : \forall^+\chi.\,\forall\alpha : \chi \to \Omega.\,\mathsf{Tran}\,(\forall\,[\chi]\,\alpha) \to \forall\alpha' : \chi.\,\mathsf{Tran}\,(\alpha\,\alpha')
\end{array}$$

For completeness in our system we also need to handle kind polymorphism and remote kind applications:

$$\begin{array}{r}
\mathsf{newkid} : \forall\alpha : (\forall\chi.\,\Omega).\,(\forall^+\chi.\,\mathsf{Tran}\,(\alpha\,[\chi])) \to \mathsf{Tran}\,(\forall^+\alpha) \\
\mathsf{rkapp} : \forall\alpha : (\forall\chi.\,\Omega).\,\mathsf{Tran}\,(\forall^+\alpha) \to \forall^+\chi.\,\mathsf{Tran}\,(\alpha\,[\chi])
\end{array}$$

Operationally, the newid's take a function between transmissible values and generate a new, globally unique identifier and tell the name server to associate that identifier with the function on the local machine. The remote applications take a proxy identifier of a remote function and a transmissible argument value. The name server is contacted to get the site where the remote value exists; the argument is sent to this machine, and the result of the function transmitted back as the result of the operation.

Marshalling and unmarshalling of values from transmissible representations are performed by the mutually recursive functions $\mathsf{M} : \forall\alpha : \Omega.\,\alpha \to \mathsf{Tran}\,\alpha$ and $\mathsf{U} : \forall\alpha : \Omega.\,\mathsf{Tran}\,\alpha \to \alpha$. They are defined below by a pattern-matching syntax and implicit recursion instead of $\mathsf{typecase}$ and fix. We assume that a type or a kind does not need to be transformed in order to be transmitted.

$$\begin{array}{lll}
\mathsf{M}\,[\mathsf{int}] & = & \lambda x : \mathsf{int}.\,x \\
\mathsf{M}\,[\alpha_1 \to \alpha_2] & = & \lambda x : \alpha_1 \to \alpha_2. \\
 & & \quad \mathsf{newid}\,[\alpha_1]\,[\alpha_2] \\
 & & \qquad (\lambda x' : \mathsf{Tran}\,\alpha_1.\,\mathsf{M}\,[\alpha_2]\,(x\,(\mathsf{U}\,[\alpha_1]\,x'))) \\
\mathsf{M}\,[\forall\,[\chi]\,\alpha] & = & \lambda x : \forall\,[\chi]\,\alpha. \\
 & & \quad \mathsf{newpid}\,[\chi]^+\,[\alpha]\,(\Lambda\alpha' : \chi.\,\mathsf{M}\,[\alpha\,\alpha']\,(x\,[\alpha'])) \\
\mathsf{M}\,[\forall^+\alpha] & = & \lambda x : \forall^+\alpha.\,\mathsf{newkid}\,[\alpha]\,(\Lambda^+\chi.\,\mathsf{M}\,[\alpha\,[\chi]]\,(x\,[\chi]^+)) \\
\mathsf{M}\,[\mathsf{Id}\,\alpha] & = & \lambda x : \mathsf{Id}\,\alpha.\,x \\
\\
\mathsf{U}\,[\mathsf{int}] & = & \lambda x : \mathsf{Tran}\,(\mathsf{int}).\,x \\
\mathsf{U}\,[\alpha_1 \to \alpha_2] & = & \lambda x : \mathsf{Tran}\,(\alpha_1 \to \alpha_2).\,\lambda x' : \alpha_1. \\
 & & \quad \mathsf{U}\,[\alpha_2]\,(\mathsf{rapp}\,[\alpha_1]\,[\alpha_2]\,x\,(\mathsf{M}\,[\alpha_1]\,x')) \\
\mathsf{U}\,[\forall\,[\chi]\,\alpha] & = & \lambda x : \mathsf{Tran}\,(\forall\,[\chi]\,\alpha).\,\Lambda\alpha' : \chi. \\
 & & \quad \mathsf{U}\,[\alpha\,\alpha']\,(\mathsf{rtapp}\,[\chi]^+\,[\alpha]\,x\,[\alpha']) \\
\mathsf{U}\,[\forall^+\alpha] & = & \lambda x : \mathsf{Tran}\,(\forall^+\alpha).\,\Lambda^+\chi.\,\mathsf{U}\,[\alpha\,[\chi]]\,(\mathsf{rkapp}\,[\alpha]\,x\,[\chi]^+) \\
\mathsf{U}\,[\mathsf{Id}\,\alpha] & = & \lambda x : \mathsf{Tran}\,(\mathsf{Id}\,\alpha).\,x
\end{array}$$

3.2 Example: Polymorphic equality

Another view at the term-level analysis of quantified types is provided by an example involving the comparison of values of existential type. The term constructs for introduction and elimination of existential types have the following formation rules.

$$\frac{\mathcal{E};\Delta;\Gamma \vdash e\ :\ (\lambda\alpha : \kappa.\,\tau)\,\tau'}{\mathcal{E};\Delta;\Gamma \vdash \langle \alpha : \kappa = \tau',\ e : \tau\rangle\ :\ \exists\alpha : \kappa.\,\tau}$$

$$\frac{\mathcal{E};\Delta;\Gamma \vdash e\ :\ \exists\,[\kappa]\,\tau \qquad \mathcal{E};\Delta \vdash \tau'\ :\ \Omega \qquad \mathcal{E};\Delta,\alpha : \kappa;\Gamma,x : \tau\,\alpha \vdash e'\ :\ \tau'}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{open}\ e\ \mathsf{as}\ \langle\alpha : \kappa,\ x : \tau\,\alpha\rangle\ \mathsf{in}\ e'\ :\ \tau'}$$

The polymorphic equality function eq is defined in Figure 7 (we use a letrec construct derived from our fix). The domain type of the function is restricted to types of the form $\mathsf{Eq}\,\tau$ to ensure that only values of types admitting equality are compared.

[1] Ohori and Kato [14] define one primitive for creating identifiers for both term and type abstraction.

$$
\begin{array}{l}
\mathsf{letrec}\\
\quad \mathsf{heq} : \forall\alpha:\Omega.\,\forall\alpha':\Omega.\,\mathsf{Eq}\,\alpha \to \mathsf{Eq}\,\alpha' \to \mathsf{Bool}\\
\quad = \Lambda\alpha:\Omega.\,\Lambda\alpha':\Omega.\\
\qquad \mathsf{typecase}[\lambda\gamma:\Omega.\,\mathsf{Eq}\,\gamma \to \mathsf{Eq}\,\alpha' \to \mathsf{Bool}]\ \alpha\ \mathsf{of}\\
\qquad\quad \mathsf{Bool} \Rightarrow \lambda \mathsf{x}:\mathsf{Bool}.\\
\qquad\qquad\qquad \mathsf{typecase}[\lambda\gamma:\Omega.\,\mathsf{Eq}\,\gamma \to \mathsf{Bool}]\ \alpha'\ \mathsf{of}\\
\qquad\qquad\qquad\quad \mathsf{Bool} \Rightarrow \lambda \mathsf{y}:\mathsf{Bool}.\ \mathsf{primEqBool}\,\mathsf{x}\,\mathsf{y}\\
\qquad\qquad\qquad\quad \ldots \Rightarrow \ldots\,\mathsf{false}\\
\qquad\quad \beta_1\times\beta_2 \Rightarrow \lambda\mathsf{x}:\mathsf{Eq}\,\beta_1\times\mathsf{Eq}\,\beta_2.\\
\qquad\qquad\qquad \mathsf{typecase}[\lambda\gamma:\Omega.\,\mathsf{Eq}\,\gamma \to \mathsf{Bool}]\ \alpha'\ \mathsf{of}\\
\qquad\qquad\qquad\quad \beta_1'\times\beta_2' \Rightarrow \lambda\mathsf{y}:\mathsf{Eq}\,\beta_1'\times\mathsf{Eq}\,\beta_2'.\\
\qquad\qquad\qquad\qquad\quad \mathsf{heq}\,[\beta_1]\,[\beta_1']\,(\mathsf{x}.1)\,(\mathsf{y}.1)\ \mathsf{and}\\
\qquad\qquad\qquad\qquad\quad \mathsf{heq}\,[\beta_2]\,[\beta_2']\,(\mathsf{x}.2)\,(\mathsf{y}.2)\\
\qquad\qquad\qquad\quad \ldots \Rightarrow \ldots\,\mathsf{false}\\
\qquad\quad \exists\,[\chi]\,\beta \Rightarrow \lambda\mathsf{x}:(\exists\beta_1:\chi.\,\mathsf{Eq}\,(\beta\,\beta_1)).\\
\qquad\qquad\qquad \mathsf{typecase}[\lambda\gamma:\Omega.\,\mathsf{Eq}\,\gamma \to \mathsf{Bool}]\ \alpha'\ \mathsf{of}\\
\qquad\qquad\qquad\quad \exists\,[\chi']\,\beta' \Rightarrow \lambda\mathsf{y}:(\exists\beta_1':\chi'.\,\mathsf{Eq}\,(\beta'\,\beta_1')).\\
\qquad\qquad\qquad\qquad\quad \mathsf{open}\ \mathsf{x}\ \mathsf{as}\ \langle\beta_1:\chi,\ \mathsf{xc}:\mathsf{Eq}\,(\beta\,\beta_1)\rangle\ \mathsf{in}\\
\qquad\qquad\qquad\qquad\quad \mathsf{open}\ \mathsf{y}\ \mathsf{as}\ \langle\beta_1':\chi',\ \mathsf{yc}:\mathsf{Eq}\,(\beta'\,\beta_1')\rangle\ \mathsf{in}\\
\qquad\qquad\qquad\qquad\qquad \mathsf{heq}\,[\beta\,\beta_1]\,[\beta'\,\beta_1']\,\mathsf{xc}\,\mathsf{yc}\\
\qquad\qquad\qquad\quad \ldots \Rightarrow \ldots\,\mathsf{false}\\
\qquad\quad \ldots\\
\mathsf{in}\ \mathsf{let}\ \mathsf{eq} = \Lambda\alpha:\Omega.\,\lambda\mathsf{x}:\mathsf{Eq}\,\alpha.\,\lambda\mathsf{y}:\mathsf{Eq}\,\alpha.\,\mathsf{heq}\,[\alpha]\,[\alpha]\,\mathsf{x}\,\mathsf{y}\\
\mathsf{in}\ \ldots
\end{array}
$$

Figure 7: Polymorphic equality in λ_i^P

Consider the two packages $v = \langle\alpha:\Omega = \mathsf{Bool},\ \mathsf{false}:\alpha\rangle$ and $v' = \langle\alpha:\Omega = \mathsf{Bool}\times\mathsf{Bool},\ \langle\mathsf{true},\ \mathsf{true}\rangle:\alpha\rangle$. Both are of type $\exists\alpha:\Omega.\,\alpha$, which makes the invocation $\mathsf{eq}\,[\exists\alpha:\Omega.\,\alpha]\,v\,v'$ legal. But when the packages are open, the types of the packaged values may (as in this example) turn out to be different. Therefore we need the auxiliary function heq to compare values of possibly different types by comparing their types first. The function corresponds to a matrix on the types of the two arguments, where the diagonal elements compare recursively the constituent values, while off-diagonal elements return false and are abbreviated in the figure.

The only interesting case is that of values of an existential type. Opening the packages provides access to the witness types β_1 and β_1' of the arguments x and y. As shown in the typing rules, the actual types of the packaged values, x and y, are obtained by applying the corresponding type functions β and β' to the respective witness types. This yields a perhaps unexpected semantics of equality. Consider this invocation of the eq function which evaluates to true:

$$
\begin{array}{l}
\mathsf{eq}\,[\exists\alpha:\Omega.\,\alpha]\\
\quad \langle\alpha:\Omega = \exists\beta:\Omega.\,\beta,\ \langle\beta:\Omega = \mathsf{Bool},\ \mathsf{true}:\mathsf{Eq}\,\beta\rangle:\mathsf{Eq}\,\alpha\rangle\\
\quad \langle\alpha:\Omega = \exists\beta:\Omega\to\Omega.\,\beta\,\mathsf{Bool},\\
\qquad\quad \langle\beta:\Omega\to\Omega = \lambda\gamma:\Omega.\,\gamma,\ \mathsf{true}:\mathsf{Eq}\,(\beta\,\mathsf{Bool})\rangle:\mathsf{Eq}\,\alpha\rangle
\end{array}
$$

At runtime, after the two packages are opened, the call to heq is

$$
\begin{array}{l}
\mathsf{heq}\,[\exists\beta:\Omega.\,\beta]\,[\exists\beta:\Omega\to\Omega.\,\beta\,\mathsf{Bool}]\\
\quad \langle\beta:\Omega = \mathsf{Bool},\ \mathsf{true}:\mathsf{Eq}\,\beta\rangle\\
\quad \langle\beta:\Omega\to\Omega = \lambda\gamma:\Omega.\,\gamma,\ \mathsf{true}:\mathsf{Eq}\,(\beta\,\mathsf{Bool})\rangle
\end{array}
$$

This term evaluates to true even though the type arguments are different. The reason is that what is being compared are the actual types of the values before hiding their witness types. Tracing the reduction of this term to the recursive call $\mathsf{heq}\,[\beta\,\beta_1]\,[\beta'\,\beta_1']\,\mathsf{xc}\,\mathsf{yc}$ we find out it is instantiated to

$$\mathsf{heq}\,[(\lambda\beta:\Omega.\,\beta)\,\mathsf{Bool}]\,[(\lambda\beta:\Omega\to\Omega.\,\beta\,\mathsf{Bool})\,(\lambda\gamma:\Omega.\,\gamma)]\,\mathsf{true}\,\mathsf{true}$$

which reduces to $\mathsf{heq}\,[\mathsf{Bool}]\,[\mathsf{Bool}]\,\mathsf{true}\,\mathsf{true}$ and thus to true.

However this result is justified, since the above two packages of type $\exists\alpha:\Omega.\,\alpha$ will indeed behave identically in all contexts. An informal argument in support of this claim is that the most any context could do with such a package is open it and inspect the type of its value using typecase, but this will only provide access to a *type function* τ representing the inner existential type. Since the kind κ of the domain of τ is unknown statically, the only non-trivial operation on τ is its application to the witness type of the package, which is the only available type of kind κ. As we saw above, this operation will produce the same result (namely Bool) in both cases. Thus, since the two arguments to eq are indistinguishable by λ_i^P contexts, the above result is perfectly sensible.

3.3 Discussion

Before we move on, it would be worthwhile to analyze the λ_i^P language. Specifically, what is the price in terms of complexity of the type theory that can be attributed to the requirements that we imposed?

In Section 2.3 we saw that an iterative type operator is essential to typechecking many type-directed operations. Even when restricted to compiling ML we still have to consider analysis of polymorphic types of the form $\forall\alpha:\Omega.\,\tau$, and their *ad hoc* inclusion in kind Ω makes the latter non-inductive. Therefore, even for this simple case, we need kind polymorphism in an essential way to handle the negative occurrence of Ω in the domain of $\forall$. In turn, kind polymorphism allows us to analyze at the type level types quantified over any kind; hence the extra expressiveness comes for free. Moreover, adding kind polymorphism does not entail any heavy type-theoretic machinery—the kind and type language of λ_i^P is a minor extension (with primitive recursion) of the well-studied calculus F_2; we use the basic techniques developed for F_2 [6] to prove properties of our type language.

The kind polymorphism of λ_i^P is parametric, *i.e.*, kind analysis is not possible. This property prevents in particular the construction of non-terminating types based on variants of Girard's J operator using a kind-comparing operator [7].

For analysis of quantified types at the term level we have the new construct $\Lambda^{+}\chi.\,e$. This does not result in any additional complexity at the type level—although we introduce a new type constructor $\forall^{+}$, the kind of this construct is defined completely by the original kind calculus, and the kind and type calculus is still essentially F_2. The term calculus becomes an extension of Girard's λU calculus [5], hence it is not normalizing; however it already includes the general recursion construct fix, necessary in a realistic programming language.

Restricting the type analysis at the term level to a finite set of kinds would help avoid the term-level kind abstraction. However, even in this case, we would still need kind abstraction to implement a type erasure semantics, which can simplify certain phases of the compiler (for details see the extended report [18]). On the other hand, having kind abstraction at the term level of λ_i^P adds no complications to the transition to type erasure semantics.

4. ANALYZING RECURSIVE TYPES

Next we turn our attention to the problem of analyzing recursive types. Following the general scheme described in the previous section, we need to introduce a type constructor μ yielding a type isomorphic to the least fixpoint of a given type function. Since the types we analyze are of kind Ω, the kind of μ of interest is

$$\mu : (\Omega\to\Omega)\to\Omega$$

Unfortunately there is a negative occurrence of Ω in the domain of this kind, which—as it was with universally-quantified types in Section 3—prevents defining an iterator over this kind while maintaining strong normalization of the type language. In the case of quantified types we were able to resolve this problem by generalizing the negative occurrence of Ω to an arbitrary kind; however such an approach is doomed in the case of recursive types since the argument of μ must have identical domain and range.

One possibility is to follow the approach outlined by Crary and Weirich in [1] for quantified types; since type variables bound by the fixpoint operator must be of kind Ω, an environment can be used to map them to types of kind Ω without kind mismatches. While plausible and perhaps efficient, this approach (as pointed out in Section 2.4) gives no protection against some programming errors, and it is unclear how to combine it with λ_i^P.

4.1 A restricted Typerec

To handle recursive types, we introduce a new constructor Place that acts as the right inverse of the Typerec. We will first give an informal explanation of how the Place constructor is used in our solution by considering a restricted form of the Typerec. This approach does not guarantee termination; we use it to ease the presentation of the λ_i^Q calculus.

Consider the iteration $\mathsf{Typerec}[\Omega]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu)$ in the case when τ is a recursive type, say $\mu\,(\lambda\alpha:\Omega.\,\mathsf{int} \rightarrow \alpha)$. In many cases, the desired result will be another recursive type, say $\mu\,(\lambda\alpha:\Omega.\,\tau')$ where τ' is the result of analyzing the body. If we followed the approach we used in the case of polymorphic types (*i.e.,* if the iterator's action on the type variable is suspended until the variable is replaced by a type upon unfolding the fixpoint), then the result would be:

$$\mu\,(\lambda\alpha:\Omega.\,\tau_{\rightarrow}\ \mathsf{int}\ \alpha\ \tau_{\mathsf{int}}\ (\mathsf{Typerec}[\Omega]\ \alpha\ \mathsf{of}\ \ldots))$$

In this case, the iterator ends up being applied n times to the nth unfolding of the fixpoint, which does not correspond to the desired fixpoint. Instead the iterator must be applied to the body of the type function, but—in contrast with the behavior in the case of a quantified type—the iterator should *disappear* when applied to the type variable α. Since the fixpoint notation represents a type isomorphic to an infinite unfolding of the body, the traversal of the entire infinite tree is complete with one iteration over the body. In other words the iterator must satisfy an equation like $\mathsf{Typerec}[\Omega]\ \alpha\ \mathsf{of}\ \ldots = \alpha$ so that the result of analyzing the body is $\lambda\alpha:\Omega.\,\tau_{\rightarrow}\ \mathsf{int}\ \alpha\ \tau_{\mathsf{int}}\ \alpha$.

Therefore, we need to distinguish between type variables bound by a polymorphic or existential quantifier and those bound in a recursive type. This reasoning leads us to a solution based on the work of Fegaras and Sheard on catamorphisms over non-inductive datatypes [4]. The main idea is to introduce an auxiliary type constructor Place of kind $\Omega \rightarrow \Omega$ which is the right inverse of the iterator, *i.e.,* it holds that

$$\mathsf{Typerec}[\Omega]\ (\mathsf{Place}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau$$

The iterator processes the body of a recursive type with the μ-bound type variable protected under Place. While processing the body, the iterator eventually reduces to instances of the form

$$\mathsf{Typerec}[\Omega]\ (\mathsf{Place}\,\alpha)\ \mathsf{of}\ \ldots,$$

which reduce to α. The reduction rule for the iterator over a recursive type is

$$\begin{array}{l}
\mathsf{Typerec}[\Omega]\ (\mu\,\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \\
\tau_\mu\,\tau' \\
\quad (\lambda\alpha:\Omega.\,\mathsf{Typerec}[\Omega]\ (\tau'\,(\mathsf{Place}\,\alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu))
\end{array}$$

$$\begin{array}{llll}
(kinds) & \kappa & ::= & \chi \mid \natural\kappa \mid \kappa \rightarrow \kappa' \mid \forall\chi.\,\kappa \\
(types) & \tau & ::= & \alpha \mid \mathsf{int} \mid \overset{\circ}{\twoheadrightarrow} \mid \mathring{\forall} \mid \mathring{\forall}^{+} \mid \mathring{\mu} \mid \mathsf{Place} \\
 & & \mid & \lambda\alpha:\kappa.\,\tau \mid \tau\,\tau' \mid \Lambda\chi.\,\tau \mid \tau\,[\kappa] \\
 & & \mid & \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \\
(values) & v & ::= & i \mid \Lambda^{+}\chi.\,v \mid \Lambda\alpha:\kappa.\,v \mid \lambda x:\tau.\,e \mid \mathsf{fix}\,x:\tau.\,v \\
 & & \mid & \mathsf{fold}\ v\ \mathsf{as}\ \tau \\
(terms) & e & ::= & v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \\
 & & \mid & \mathsf{fold}\ e\ \mathsf{as}\ \tau \mid \mathsf{unfold}\ e\ \mathsf{as}\ \tau \\
 & & \mid & \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+};\ e_\mu)
\end{array}$$

Figure 8: The λ_i^Q language

$$\begin{array}{rl}
\Omega & \equiv \forall\chi.\,\natural\chi \\
\tau\$\tau' & \equiv \Lambda\chi.\,\tau\,[\chi]\,(\tau'\,[\chi]) \quad \text{for } \chi \notin \mathit{fkv}(\tau) \cup \mathit{fkv}(\tau') \\
\tau \rightarrow \tau' & \equiv (\twoheadrightarrow)\,\tau\,\tau' \\
\forall\alpha:\kappa.\,\tau & \equiv \boldsymbol{\forall}\,[\kappa]\,(\lambda\alpha:\kappa.\,\tau) \\
\forall^{+}\chi.\,\tau & \equiv \boldsymbol{\forall}^{+}(\Lambda\chi.\,\tau)
\end{array}$$

$$\begin{array}{lll}
(\twoheadrightarrow) : \Omega \rightarrow \Omega \rightarrow \Omega & = & \lambda\alpha:\Omega.\,\lambda\alpha':\Omega.\,((\overset{\circ}{\twoheadrightarrow})\$\alpha)\$\alpha' \\
\boldsymbol{\forall} \ : \forall\chi.\,(\chi \rightarrow \Omega) \rightarrow \Omega & = & \Lambda\chi.\,\lambda\alpha:\chi \rightarrow \Omega.\,\Lambda\chi'. \\
 & & \quad \mathring{\forall}\,[\chi']\,[\chi]\,(\lambda\alpha':\chi.\,\alpha\,\alpha'\,[\chi']) \\
\boldsymbol{\forall}^{+} : (\forall\chi.\,\Omega) \rightarrow \Omega & = & \lambda\alpha:(\forall\chi.\,\Omega).\,\Lambda\chi'. \\
 & & \quad \mathring{\forall}^{+}[\chi']\,(\Lambda\chi.\,\alpha\,[\chi]\,[\chi']) \\
\mu \ : (\forall\chi.\,\natural\chi \rightarrow \natural\chi) \rightarrow \Omega & = & \lambda\alpha:(\forall\chi.\,\natural\chi \rightarrow \natural\chi).\,\mathring{\mu}\$\alpha
\end{array}$$

Figure 9: Syntactic sugar for λ_i^Q

4.2 The general case

The previous approach does not generalize to the case when the result of the Typerec may be of an arbitrary kind. In the general case, the type reductions are:

$$\begin{array}{l}
\mathsf{Typerec}[\kappa]\ (\mathsf{Place}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau \\
\mathsf{Typerec}[\kappa]\ (\mu\,\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \\
\tau_\mu\,\tau' \\
\quad (\lambda\alpha:\kappa.\,\mathsf{Typerec}[\kappa]\ (\tau'\,(\mathsf{Place}\,\alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu))
\end{array}$$

The constructor Place can now be applied to a type of arbitrary kind, but its return result must be Ω. This implies that Place has the kind $\forall\chi.\,\chi \rightarrow \Omega$. But this is unsound since we can not constrain the kind of τ above (the argument of Place) to match the result kind κ of the Typerec.

Adopting the solution given by Fegaras and Sheard, we modify the domain of intensional analysis: in place of Ω we introduce a parameterized kind $\natural$, and require that the type τ being analyzed in $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu)$ is of kind $\natural\kappa$. The constructor Place must then have the polymorphic kind $\forall\chi.\,\chi \rightarrow \natural\chi$, and the fix-point constructor $\mathring{\mu}$ the kind $\forall\chi.\,(\natural\chi \rightarrow \natural\chi) \rightarrow \natural\chi$.

We define the λ_i^Q calculus in Figures 8 and 9. Figures 10, 11,

$\boxed{\text{Kind formation} \quad \mathcal{E} \vdash \kappa}$

$$\frac{\chi \in \mathcal{E}}{\mathcal{E} \vdash \chi} \qquad \frac{\mathcal{E} \vdash \kappa}{\mathcal{E} \vdash \natural\kappa} \qquad \frac{\mathcal{E} \vdash \kappa_1 \quad \mathcal{E} \vdash \kappa_2}{\mathcal{E} \vdash \kappa_1 \to \kappa_2} \qquad \frac{\mathcal{E}, \chi \vdash \kappa}{\mathcal{E} \vdash \forall\chi.\,\kappa}$$

$\boxed{\text{Type formation} \quad \mathcal{E};\Delta \vdash \tau : \kappa}$

$$\frac{\mathcal{E} \vdash \Delta \quad \alpha{:}\kappa \text{ in } \Delta}{\mathcal{E};\Delta \vdash \alpha : \kappa} \qquad \frac{\mathcal{E} \vdash \Delta}{\begin{array}{ll} \mathcal{E};\Delta \vdash \mathsf{int} & : \forall\chi.\,\natural\chi \\ \mathcal{E};\Delta \vdash (\overset{\circ}{\twoheadrightarrow}) & : \forall\chi.\,\natural\chi \to \natural\chi \to \natural\chi \\ \mathcal{E};\Delta \vdash \mathring{\forall} & : \forall\chi.\,\forall\chi'.\,(\chi' \to \natural\chi) \to \natural\chi \\ \mathcal{E};\Delta \vdash \mathring{\forall}^{+} & : \forall\chi.\,(\forall\chi'.\,\natural\chi) \to \natural\chi \\ \mathcal{E};\Delta \vdash \mathring{\mu} & : \forall\chi.\,(\natural\chi \to \natural\chi) \to \natural\chi \\ \mathcal{E};\Delta \vdash \mathsf{Place} & : \forall\chi.\,\chi \to \natural\chi \end{array}}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa \vdash \tau : \kappa'}{\mathcal{E};\Delta \vdash \lambda\alpha{:}\kappa.\,\tau : \kappa \to \kappa'} \qquad \frac{\mathcal{E};\Delta \vdash \tau : \kappa' \to \kappa \quad \mathcal{E};\Delta \vdash \tau' : \kappa'}{\mathcal{E};\Delta \vdash \tau\,\tau' : \kappa}$$

$$\frac{\mathcal{E},\chi;\Delta \vdash \tau : \kappa}{\mathcal{E};\Delta \vdash \Lambda\chi.\,\tau : \forall\chi.\,\kappa} \qquad \frac{\mathcal{E};\Delta \vdash \tau : \forall\chi.\,\kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E};\Delta \vdash \tau\,[\kappa'] : \kappa\{\kappa'/\chi\}}$$

$$\frac{\begin{array}{ll} \mathcal{E};\Delta \vdash \tau & : \natural\kappa \\ \mathcal{E};\Delta \vdash \tau_{\mathsf{int}} & : \kappa \\ \mathcal{E};\Delta \vdash \tau_{\to} & : \natural\kappa \to \natural\kappa \to \kappa \to \kappa \to \kappa \\ \mathcal{E};\Delta \vdash \tau_{\forall} & : \forall\chi.\,(\chi \to \natural\kappa) \to (\chi \to \kappa) \to \kappa \\ \mathcal{E};\Delta \vdash \tau_{\forall^+} & : (\forall\chi.\,\natural\kappa) \to (\forall\chi.\,\kappa) \to \kappa \\ \mathcal{E};\Delta \vdash \tau_{\mu} & : (\natural\kappa \to \natural\kappa) \to (\kappa \to \kappa) \to \kappa \end{array}}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{\mu}) : \kappa}$$

Figure 10: λ_i^Q type formation rules

$\boxed{\text{Term formation} \quad \mathcal{E};\Delta;\Gamma \vdash e : \tau}$

$$\frac{\mathcal{E};\Delta \vdash \Gamma}{\mathcal{E};\Delta;\Gamma \vdash i : \mathsf{int}} \qquad \frac{\mathcal{E};\Delta;\Gamma \vdash e : \tau \quad \mathcal{E};\Delta \vdash \tau \leadsto \tau' : \Omega}{\mathcal{E};\Delta;\Gamma \vdash e : \tau'}$$

$$\frac{\mathcal{E};\Delta \vdash \tau : \forall\chi.\,\natural\chi \to \natural\chi \quad \mathcal{E};\Delta;\Gamma \vdash e : \mu\tau}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{unfold}\ e\ \mathsf{as}\ \tau : \tau\$(\mu\tau)}$$

$$\frac{\mathcal{E};\Delta \vdash \tau : \forall\chi.\,\natural\chi \to \natural\chi \quad \mathcal{E};\Delta;\Gamma \vdash e : \tau\$(\mu\tau)}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{fold}\ e\ \mathsf{as}\ \tau : \mu\tau}$$

$$\frac{\mathcal{E},\chi;\Delta;\Gamma \vdash v : \tau}{\mathcal{E};\Delta;\Gamma \vdash \Lambda^{+}\chi.\,v : \forall^{+}\chi.\,\tau} \qquad \frac{\mathcal{E};\Delta;\Gamma \vdash e : \boldsymbol{\forall}^{+}\tau \quad \mathcal{E} \vdash \kappa}{\mathcal{E};\Delta;\Gamma \vdash e\,[\kappa]^{+} : \tau\,[\kappa]}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa;\Gamma \vdash e : \tau}{\mathcal{E};\Delta;\Gamma \vdash \Lambda\alpha{:}\kappa.\,e : \forall\alpha{:}\kappa.\,\tau} \qquad \frac{\mathcal{E};\Delta;\Gamma,x{:}\tau \vdash e : \tau'}{\mathcal{E};\Delta;\Gamma \vdash \lambda x{:}\tau.\,e : \tau \to \tau'}$$

$$\frac{\mathcal{E};\Delta;\Gamma \vdash e : \boldsymbol{\forall}\,[\kappa]\,\tau \quad \mathcal{E};\Delta \vdash \tau' : \kappa}{\mathcal{E};\Delta;\Gamma \vdash e\,[\tau'] : \tau\,\tau'}$$

$$\frac{\mathcal{E};\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \mathcal{E};\Delta;\Gamma \vdash e_2 : \tau_2}{\mathcal{E};\Delta;\Gamma \vdash e_1\,e_2 : \tau_1}$$

$$\frac{\begin{array}{c} \mathcal{E};\Delta;\Gamma,x{:}\tau \vdash v : \tau \\ \tau = \forall^{+}\chi_1 \ldots \chi_n.\,\forall\alpha_1{:}\kappa_1 \ldots \alpha_m{:}\kappa_m{:}\tau_1 \to \tau_2. \\ n \geq 0, m \geq 0 \end{array}}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{fix}\,x{:}\tau.\,v : \tau}$$

$$\frac{\begin{array}{ll} \mathcal{E};\Delta \vdash \tau & : \Omega \to \Omega \\ \mathcal{E};\Delta \vdash \tau' & : \Omega \\ \mathcal{E};\Delta;\Gamma \vdash e_{\mathsf{int}} & : \tau\,\mathsf{int} \\ \mathcal{E};\Delta;\Gamma \vdash e_{\to} & : \forall\alpha{:}\Omega.\,\forall\alpha'{:}\Omega.\,\tau\,(\alpha_1 \to \alpha_2) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\forall} & : \forall^{+}\chi.\,\forall\alpha{:}\chi \to \Omega.\,\tau\,(\boldsymbol{\forall}\,[\chi]\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\forall^+} & : \forall\alpha{:}(\forall\chi.\,\Omega).\,\tau\,(\boldsymbol{\forall}^{+}\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\mu} & : \forall\alpha{:}(\forall\chi.\,\natural\chi \to \natural\chi).\,\tau\,(\mu\,\alpha) \end{array}}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) : \tau\,\tau'}$$

Figure 11: λ_i^Q term formation rules

and 12 show the static semantics. Figure 13 shows the dynamic semantics.

Types which had kind Ω in λ_i^P could be analyzed by a Typerec with an arbitrary result kind κ'. In our new language λ_i^Q, a type that can be analyzed by an arbitrary Typerec construct must have the kind $\natural\kappa$ for all possible κ. Thus the kind Ω of λ_i^P is represented by the kind $\forall\chi.\,\natural\chi$ in λ_i^Q.

To be able to analyze function and polymorphic types, we now have to modify their kinds as well; to avoid confusion with the constructors based on Ω, we denote the new constructors by $\overset{\circ}{\twoheadrightarrow}$, $\mathring{\forall}$, and $\mathring{\forall}^{+}$ (Figure 8). The kind rules for these constructors are shown in Figure 10. We can define equivalents of the λ_i^P types $(\twoheadrightarrow)$, $\boldsymbol{\forall}$, and $\boldsymbol{\forall}^{+}$ starting from $\overset{\circ}{\twoheadrightarrow}$, $\mathring{\forall}$, and $\mathring{\forall}^{+}$ respectively. The key intuition in the definition (Figure 9) is that we thread the same kind through all components of kind Ω. For example, expanding the definition of $\tau \to \tau'$ we obtain its equivalent, $\Lambda\chi.\,\overset{\circ}{\twoheadrightarrow}[\chi]\,(\tau\,[\chi])(\tau'\,[\chi])$. Expressed in terms of these derived types, the typing rules for most λ_i^Q terms (Figure 11) are identical to those of λ_i^P. Compared with λ_i^P, the term language of λ_i^Q has two new constructs – $\mathsf{fold}\ e\ \mathsf{as}\ \tau$ and $\mathsf{unfold}\ e\ \mathsf{as}\ \tau$ – to implement the isomorphism between a recursive type and its unfolding.

Each of these constructors must first be applied to kind κ before being analyzed, where κ is the kind of the result of the analysis. In all other aspects the type-level analysis proceeds as in λ_i^P by iterating over the components of the type and then passing the results of the iteration and the original components to the corresponding branch of the iterator. For example, consider the analysis of the int and $\mathring{\forall}$ constructors (Figure 12) :

$$\begin{array}{l} \mathsf{Typerec}[\kappa]\ (\mathsf{int}\,[\kappa])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{\mu}) \leadsto \tau_{\mathsf{int}} \\ \mathsf{Typerec}[\kappa]\ (\mathring{\forall}\,[\kappa]\,[\kappa']\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{\mu}) \leadsto \\ \tau_{\forall}\,[\kappa']\,\tau\,(\lambda\alpha{:}\kappa'.\,\mathsf{Typerec}[\kappa]\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{\mu})) \end{array}$$

The reduction rules for typecase are similar to those in λ_i^P, with the recursive type handled in an obvious way (Figure 13). However, there is one subtlety in the typecase reduction rules. Since typecase does not iterate over the structure of a type, its reductions do not introduce the Place constructor; thus the type analyzed by Typerec$[\kappa]$ must be of kind $\natural\kappa$, but a typecase can only analyze types of kind Ω, *i.e.*, $\forall\chi.\,\natural\chi$. It is easy to see that there are no closed types of this kind constructed using Place. Thus there are no reduction rules for typecase analyzing the Place constructor. We show this (in the companion technical report [18]) when proving the soundness of λ_i^Q.

Type reduction $\quad \mathcal{E};\Delta \vdash \tau_1 \leadsto \tau_2 : \kappa$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa' \vdash \tau : \kappa \qquad \mathcal{E};\Delta \vdash \tau' : \kappa'}{\mathcal{E};\Delta \vdash (\lambda\alpha{:}\kappa'.\,\tau)\,\tau' \leadsto \tau\{\tau'/\alpha\} : \kappa}$$

$$\frac{\mathcal{E},\chi;\Delta \vdash \tau : \forall\chi.\,\kappa \qquad \mathcal{E} \vdash \kappa'}{\mathcal{E};\Delta \vdash (\Lambda\chi.\,\tau)\,[\kappa'] \leadsto \tau\{\kappa'/\chi\} : \kappa\{\kappa'/\chi\}}$$

$$\frac{\mathcal{E};\Delta \vdash \tau : \kappa \to \kappa' \qquad \alpha \notin ftv(\tau)}{\mathcal{E};\Delta \vdash \lambda\alpha{:}\kappa.\,\tau\,\alpha \leadsto \tau : \kappa \to \kappa'}$$

$$\frac{\mathcal{E};\Delta \vdash \tau : \forall\chi'.\,\kappa \qquad \chi \notin fkv(\tau)}{\mathcal{E};\Delta \vdash \Lambda\chi.\,\tau\,[\chi] \leadsto \tau : \forall\chi'.\,\kappa}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathsf{int}\,[\kappa])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) : \kappa}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathsf{int}\,[\kappa])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_{\mathsf{int}} : \kappa}$$

$$\frac{\begin{array}{c}\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_1' : \kappa \\ \mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_2' : \kappa\end{array}}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ ((\mathring{\twoheadrightarrow})\,[\kappa]\,\tau_1\,\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_{\to}\,\tau_1\,\tau_2\,\tau_1'\,\tau_2' : \kappa}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa' \vdash \mathsf{Typerec}[\kappa]\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau' : \kappa}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathring{\forall}\,[\kappa]\,[\kappa']\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_{\forall}\,[\kappa']\,\tau\,(\lambda\alpha{:}\kappa'.\,\tau') : \kappa}$$

$$\frac{\mathcal{E},\chi;\Delta \vdash \mathsf{Typerec}[\kappa]\ (\tau\,[\chi])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau' : \kappa}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathring{\forall}^{+}[\kappa]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_{\forall^+}\,\tau\,(\Lambda\chi.\,\tau') : \kappa}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa \vdash \mathsf{Typerec}[\kappa]\ (\tau\,(\mathsf{Place}\,[\kappa]\,\alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau' : \kappa}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathring{\mu}\,[\kappa]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau_\mu\,\tau\,(\lambda\alpha{:}\kappa.\,\tau') : \kappa}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathsf{Place}\,[\kappa]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) : \kappa}{\mathcal{E};\Delta \vdash \mathsf{Typerec}[\kappa]\ (\mathsf{Place}\,[\kappa]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_\mu) \leadsto \tau : \kappa}$$

Figure 12: Selected λ_i^Q type reduction rules

$$\mathsf{unfold}\ (\mathsf{fold}\ v\ \mathsf{as}\ \tau)\ \mathsf{as}\ \tau \leadsto v$$

$$\frac{e \leadsto e'}{\mathsf{fold}\ e\ \mathsf{as}\ \tau \leadsto \mathsf{fold}\ e'\ \mathsf{as}\ \tau} \qquad \frac{e \leadsto e'}{\mathsf{unfold}\ e\ \mathsf{as}\ \tau \leadsto \mathsf{unfold}\ e'\ \mathsf{as}\ \tau}$$

$$\mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu) \leadsto e_{\mathsf{int}}$$

$$\mathsf{typecase}[\tau]\ (\tau_1 \to \tau_2)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu) \leadsto e_{\to}\,[\tau_1]\,[\tau_2]$$

$$\mathsf{typecase}[\tau]\ (\forall\,[\kappa]\,\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu) \leadsto e_{\forall}\,[\kappa]^{+}\,[\tau']$$

$$\mathsf{typecase}[\tau]\ (\forall^{+}\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu) \leadsto e_{\forall^+}\,[\tau']$$

$$\mathsf{typecase}[\tau]\ (\mu\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu) \leadsto e_\mu\,[\tau']$$

$$\frac{\varepsilon;\varepsilon \vdash \tau' \leadsto^{*} \nu'{:}\Omega \qquad \nu' \text{ is a normal form}}{\begin{array}{l}\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu) \leadsto \\ \qquad \mathsf{typecase}[\tau]\ \nu'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_\mu)\end{array}}$$

Figure 13: Selected λ_i^Q term reduction rules

The language λ_i^Q enjoys the properties of λ_i^P listed in Section 3, detailed proofs of which can be found in the companion technical report [18]. For instance, we prove strong normalization using Girard's method of candidates [6] as for λ_i^P, with a few adjustments: Since our "base" kind $\natural$ is parametric, we define $R_\natural\mathcal{C}_\kappa$ as the set of types τ of kind $\natural\kappa$ for which $\mathsf{Typerec}[\kappa]\,\tau \ldots$ belongs to a candidate $\mathcal{C}_\kappa$ of kind κ whenever the branches belong to candidates of the respective kinds, and the set $\mathcal{S}_{\natural\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ is defined as $R_\natural(\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}])$.

4.3 Limitations

The approach outlined in this section allows the analysis of recursive types within the term language and the type language, but imposes severe limitations on combining these analyses. While one can write a polymorphic equality function of type $\forall\alpha:\Omega.\,\alpha \to \alpha \to \mathsf{Bool}$, and one can write a type operator Eq as in Section 3, it is not possible to write polymorphic equality of type $\forall\alpha{:}\Omega.\ \mathsf{Eq}\,\alpha \to \mathsf{Eq}\,\alpha \to \mathsf{Bool}$. The reason is that although $\mathsf{Eq}\,(\mu\,\tau)$ reduces to a recursive type, its unfolding is not $\mathsf{Eq}\,(\tau\$(\mu\,\tau))$, the type needed for the recursive invocation of the equality function. Indeed the types $\tau'\,(\mu\,\tau)$ and $\tau'\,(\tau\$(\mu\,\tau))$ are not bisimilar in general, since τ' may analyze its argument and produce different results depending on whether it is a recursive type or not. Thus the problem can be traced back to our decision to define $\mathring{\mu}$ as a "constructor" for kind $\natural$, which makes recursive types observably distinct from their unfoldings. Alternatives are to limit the result kind of $\mathsf{Typerec}$ to Ω, or to regain transparency of $\mathring{\mu}$ by eliminating the τ_μ branch of $\mathsf{Typerec}$ and providing a reduction rule which always maps recursive types to recursive types; since the analogous transformation at the term level in the latter case will require combining $\mathsf{typecase}$ with recursion, the resulting language exceeds the scope of the current paper.

5. RELATED WORK

The work of Harper and Morrisett [8] introduced intensional type analysis and pointed out the necessity for type-level type analysis operators which inductively traverse the structure of types. The domain of their analysis is restricted to a predicative subset of the type language, which prevents its use in programs which must support all types of values, including polymorphic functions, closures, and objects. This paper builds on their work by extending type analysis to include the full type language. Crary *et al.* [1] propose a very powerful type analysis framework. They define a rich kind calculus that includes sum kinds and inductive kinds. They also provide primitive recursion at the type level. Therefore, they can define new kinds within their calculus and directly encode type analysis operators within their language. They also include a novel refinement operation at the term level. However, their type analysis is "limited to parametrically polymorphic functions, and cannot account for functions that perform intensional type analysis" [1, Section 4.1].

Our type analysis can also handle polymorphic functions that analyze the quantified type variable. Moreover, their type analysis is not fully reflexive since they can not handle arbitrary quantified types; quantification must be restricted to type variables of kind Ω. Duggan [3] proposes another framework for intensional type analysis; however, he allows the analysis of types only at the term level and not at the type level. Yang [27] presents some approaches to enable type-safe programming of type-indexed values in ML which is similar to term-level analysis of types. Our solution for recursive types is based on the idea proposed by Fegaras and Sheard [4] for extending the fold operation to non-inductive datatypes. Meijer and Hutton [10] also propose a method for extending catamorphisms to datatypes with embedded functions; however, their method requires the definition of an anamorphism for every such catamorphism.

Necula [13] proposed the ideas of a certifying compiler and implemented a certifying compiler for a type-safe subset of C. Morrisett *et al.* [12] showed that a fully type-preserving compiler generating type-safe assembly code is a practical basis for a certifying compiler.

The idea of programming with iterators is explained in Pierce's notes [16]. Pfenning and Mohring [15] show how inductively defined types can be represented by closed types. They also construct representations of all primitive recursive functions over inductively defined types.

6. CONCLUSIONS

We presented a type-theoretic framework for fully reflexive intensional analysis of types which includes analysis of polymorphic, existential, and recursive types. We can analyze arbitrary types both at the type level and at the term level. Moreover, we are not restricted to analyzing only parametrically polymorphic functions; we can also handle polymorphic functions that analyze the quantified type variable. We proved the calculus sound and showed that type checking still remains decidable. Since we can analyze arbitrary types, we can now use these constructs to write type-dependent runtime services that can operate on values of any type; as an example we showed how to use reflexive type analysis to support type-safe marshalling.

Acknowledgments

We are grateful to the anonymous referees for their insightful comments and suggestions on improving the presentation.

REFERENCES

[1] K. Crary and S. Weirich. Flexible type analysis. In *Proc. 1999 ACM SIGPLAN International Conf. on Functional Programming*, pages 233–248. ACM Press, Sept. 1999.

[2] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. 1998 ACM SIGPLAN International Conf. on Functional Programming*, pages 301–312. ACM Press, Sept. 1998.

[3] D. Duggan. A type-based semantics for user-defined marshalling in polymorphic languages. In X. Leroy and A. Ohori, editors, *Proc. 1998 International Workshop on Types in Compilation*, volume 1473 of *LNCS*, pages 273–298, Kyoto, Japan, Mar. 1998. Springer-Verlag.

[4] L. Fegaras and T. Sheard. Revisiting catamorphism over datatypes with embedded functions. In *23rd Annual ACM Symp. on Principles of Programming Languages*, pages 284–294. ACM Press, Jan. 1996.

[5] J. Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, University of Paris VII, 1972.

[6] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[7] R. Harper and J. C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, April 1999.

[8] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM Symp. on Principles of Programming Languages*, pages 130–141. ACM Press, Jan. 1995.

[9] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *Proc. 1999 ACM SIGPLAN International Conf. on Functional Programming (ICFP'99)*, pages 183–196. ACM Press, September 1999.

[10] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Functional Programming and Computer Architecture*, 1995.

[11] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd Annual ACM Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[12] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th Annual ACM Symp. on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[13] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.

[14] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Proc. 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 99–112. ACM Press, 1993.

[15] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Proc. Fifth Conf. on the Mathematical Foundations of Programming Semantics*, pages 209–228, New Orleans, Louisiana, Mar. 1989. Springer-Verlag.

[16] B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, 1989.

[17] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[18] B. Saha, V. Trifonov, and Z. Shao. Fully reflexive intensional type analysis. Technical Report YALEU/DCS/TR-1194, Dept. of Computer Science, Yale University, New Haven, CT, March 2000. Available at URL `flint.cs.yale.edu/flint/publications`.

[19] Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conf. on Functional Programming*, pages 85–98. ACM Press, June 1997.

[20] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[21] Z. Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conf. on Functional Programming*. ACM Press, 1998.

[22] Z. Shao. Transparent modules with fully syntactic signatures. In *Proc. 1999 ACM SIGPLAN International Conf. on Functional Programming (ICFP'99)*, pages 220–232. ACM Press, September 1999.

[23] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 116–129, New York, 1995. ACM Press.

[24] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Dec. 1996. Tech Report CMU-CS-97-108.

[25] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 181–192. ACM Press, 1996.

[26] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical report, Dept. of Computer Science, Rice University, June 1992.

[27] Z. Yang. Encoding types in ML-like languages. In *Proc. 1998 ACM SIGPLAN International Conf. on Functional Programming*, pages 289–300. ACM Press, 1998.