Intensional Analysis of Quantified Types

BRATIN SAHA, VALERY TRIFONOV, and ZHONG SHAO
Yale University

Compilers for polymorphic languages can use run-time 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 quantified types such as existential or polymorphic types. This makes it impossible to code (in a type-safe language) applications such as garbage collection, persistency, or marshalling which must be able to examine the type of any run-time value.

We present a typed intermediate language that supports the analysis of quantified types. 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 also show that our system is compatible with a type-erasure semantics.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.3 [**Logic and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms: Languages, Verification

Additional Key Words and Phrases: Certified Code, Runtime Type Dispatch, Typed Intermediate Languages, Intensional Type Analysis

1. INTRODUCTION

Run-time type analysis is used extensively in various applications and programming situations. Run-time services such as garbage collection, dynamic linking, and reflection, applications such as marshalling and pickling, type-safe persistent programming, and unboxing implementations of polymorphic languages all analyze types at run time. Most existing compilers use untyped intermediate languages for compilation; therefore, they support run-time type inspection in a type-unsafe manner. In this paper, we present a statically typed intermediate language that allows run-time 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 run-time type analysis in a type-safe manner has been an active area of research. This paper builds on existing work [Harper and Morrisett

This work was sponsored in part by DARPA OASIS grant F30602-99-1-0519, NSF grants CCR-9633390, CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies. Authors' addresses: Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520 USA; email: {saha, trifonov, shao}@cs.yale.edu.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

1995; Crary et al. 1998] and makes several important new contributions. We show how to support analysis of all well-formed quantified types, with bound variables ranging over arbitrary kinds, at both the term level and the type level. At the term level this enables programs to analyze run-time values such as function closures and polymorphic data structures. At the type level, type analysis provides accurate specifications for term-level type-analyzing programs. In addition, type transformations (such as those performed during closure conversion and CPS conversion), that could hitherto be expressed only in a meta language, can now be expressed within the type language itself [Shao et al. 2002]. We prove that the language is sound and that type reduction is strongly normalizing and confluent. Finally, we provide a translation to a language with type erasure semantics.

The rest of this paper is organized as follows. In Section 2 we argue the case for intensional analysis of quantified types, and describe the obstacles on the possible roads to it. Section 3 introduces our intensional polymorphic lambda calculus λ_i^ω, equipped with polymorphic kinds, which allow us to make use of kind parametricity at the type level in order to restore the inductive structure of the base kind. We also present some semantic properties of λ_i^ω and examples of its applications in Section 3. A possible path for implementing λ_i^ω is outlined in Section 4, where we develop a language for intensional type analysis of quantified types which has type-erasure semantics, and in Section 5, where we show how to translate λ_i^ω terms into it. A brief review of the related work can be found in Section 6, and proofs of the properties of λ_i^ω are included in the Appendix.

2. MOTIVATION AND APPROACH

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

—Java adopts dynamic linking as a key feature. To ensure safe linking, an external module must be dynamically verified to satisfy the expected interface type.

—A precise garbage collector must keep track of all live heap objects, and for that type information must be kept at run time 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.

—Finally, in polymorphic languages like ML, the type of a value may not be known statically; therefore, compilers have traditionally used inefficient, uniformly boxed representation. To avoid this, several modern compilers [Shao and Appel 1995; Shao 1997a; Tarditi et al. 1996] use run-time type information to support unboxed representation.

Most existing compilers use an untyped intermediate language for compiling code that involves run-time type inspection. They reify types into values and discard type information at some early stage during compilation. However, this approach is infeasible in a certifying compiler [Necula 1998].

Code certification is appealing for a number of reasons. In a certifying framework, one need not trust the correctness of the compiler that generated the certified code; instead, one can verify that the generated code satisfies the properties it claims, for instance type safety, or a specific security policy. Checking the correctness of a compiler-generated proof (of a program property) is much easier than proving the correctness of the compiler. Furthermore, 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. In this context, the compiler may not even be known to the client, and trusting it is not sufficient either—the client must now also trust the medium. For such a system to be practical, a client should be able to accept code from untrusted sources, but have a means of verifying its behavior 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 uses only granted resources, makes legal function calls, etc. Generated code which performs run-time type analysis must also be verifiable in this type system.

Moreover, type-safe run-time type analysis is also required for type-safe implementations of runtime services. The safety of a mobile code system depends not only on the downloaded code, but also on the safety of all the applications and services that the runtime system provides (since the downloaded code may execute these applications). These include services such as garbage collection, linking, etc. Typically, this code constitutes the trusted computing base of the system—it is assumed that the code is correct. However, there are significant advantages to independently verifying these runtime services. Lifting these services out of the trusted computing base makes the system more reliable. The services can be then structured as libraries, offering opportunities for code reuse.

Finally, it is essential to support analysis of quantified types. Most type-analyzing applications must handle arbitrary heap values. For example, a garbage collector needs to traverse all live data structures in the heap. In a type-preserving compiler, a closure would have an existential type [Minamide et al. 1996] and a polymorphic function would have a polymorphic type. Thus analysis of quantified types is essential in supporting these applications.

2.1 Background

Harper and Morrisett [1995] proposed intensional type analysis and presented a type-theoretic framework for expressing computations that analyze types at run time. They considered a language with operators for type analysis, both at the term level and at the type level. Type-dependent primitive functions use these operators to analyze types and select the appropriate code. For example, suppose that arrays of values of type int and real have specialized representations (with types, say, intarray and realarray), and are therefore accessed using special subscript functions intsub and realsub, while arrays of elements of any other type τ have the default boxed representation, have type $\mathsf{boxedarray}\,\tau$, and are subscripted using $\mathsf{boxedsub}\,[\tau]$. A polymorphic subscript function for arrays might be written using a term-level type analysis operator typecase as the following pseudo-code:

$$\begin{aligned}\mathsf{sub} = \Lambda\alpha.\,\mathsf{typecase}\;\alpha\;\mathsf{of}&\\ \mathsf{int}\;&\Rightarrow \mathsf{intsub}\\ \mathsf{real}\;&\Rightarrow \mathsf{realsub}\\ \beta\;&\Rightarrow \mathsf{boxedsub}\,[\beta]\end{aligned}$$

Thus sub analyzes the type α of the array elements and returns the appropriate subscript function.

Finding a type for this subscript function is more interesting, because it can be instantiated to have any one of the types $\mathsf{intarray} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{realarray} \to \mathsf{int} \to \mathsf{real}$, and $\forall\alpha.\,\mathsf{boxedarray}\,\alpha \to \mathsf{int} \to \alpha$. Since the type of an instance of sub depends on the type argument, in order to assign a type to the function we need a type-level construct, say Typecase, 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

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

The Typecase construct in the above example is a special case of the Typerec construct of Harper and Morrisett [1995], which supports primitive recursion over the monotypes (type constructors) of their language λ_i^{ML}. Their term language cannot express general recursion, either, and is also equipped with a construct for 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 or existential types). Therefore, type analyzing primitives that handle polymorphic code blocks, or closures, 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

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

Fig. 1. The type language of Harper and Morrisett

lambda calculus, with no polymorphic types. The Typerec operator analyzes only constructors of the base kind Ω:

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

The kinds of the *arguments* of these constructors do not contain any negative occurrences of the kind Ω (that is, occurrences to the left of an odd number of arrows). Thus the kind Ω is inductive. The Typerec operator provides a form of primitive recursion over this inductively defined set of types. Each instance of Typerec must specify the result of the analysis in the case of the nullary constructor int, as well as an operator to combine the subterms τ_1 and τ_2 of a function type $\tau_1 \to \tau_2$ and the results of the iteration over them. The reduction rules for Typerec can be written as

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

Operationally, the reduction of Typerec examines the head constructor of the type being analyzed and chooses a branch accordingly. If the constructor is int, the type reduces to the τ_{int} branch. If the constructor is of the form $\tau_1 \to \tau_2$, the analysis proceeds recursively on its subterms τ_1 and τ_2. The Typerec operator then applies the $\tau_{\to}$ branch to the components τ_1 and τ_2, and to the result of the iteration over these components.

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

$$
\mathsf{Typerec}\ (\forall_\Omega\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}) \ \leadsto\ \tau_{\forall}\,\tau\,(\lambda\alpha{:}\Omega.\,\mathsf{Typerec}\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall})).
$$

However the negative occurrence of Ω in the kind of the argument of $\forall_\Omega$ poses a problem: this iterator may fail to terminate! Consider the following example: Assuming $I \equiv \lambda\alpha : \Omega.\,\alpha$ and $\tau_{\forall} \equiv \lambda\beta_1 : \Omega \to \Omega.\,\lambda\beta_2{:}\Omega \to \Omega.\,\beta_2\,(\forall_\Omega\,\beta_1)$, the following reduction sequence will go on indefinitely:

$$
\begin{array}{ll}
 & \mathsf{Typerec}\ (\forall_\Omega\,I)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}) \\
\leadsto & \tau_{\forall}\,I\,(\lambda\alpha{:}\Omega.\,\mathsf{Typerec}\ (I\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall})) \\
\leadsto^3 & \mathsf{Typerec}\ (I\,(\forall_\Omega\,I))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}) \\
\leadsto & \mathsf{Typerec}\ (\forall_\Omega\,I)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}) \\
\leadsto & \ldots
\end{array}
$$

Clearly this makes the standard method of typechecking (by comparing normal forms of types) fail. More generally, the existence of an injection $\forall_\Omega$ from $\Omega \to \Omega$ to Ω, and projections from Ω to $\Omega \to \Omega$, for instance

$$\begin{aligned}\mathsf{A} \equiv \lambda\alpha\!:\!\Omega.\ \mathsf{Typerec}\ \alpha\ \mathsf{of}\ (I;\ & \\ & \lambda_\!:\!\Omega.\ \lambda_\!:\!\Omega \to \Omega.\ \lambda_\!:\!\Omega.\ \lambda_\!:\!\Omega \to \Omega.\ I; \\ & \lambda\alpha'\!:\!\Omega \to \Omega \to \Omega.\ \lambda_\!:\!\Omega \to \Omega.\ \alpha'),\end{aligned}$$

such that $\mathsf{A}\,(\forall_\Omega\,\tau) = \tau$ for all τ of kind $\Omega \to \Omega$, means that for every term of the untyped lambda calculus one can construct a corresponding well-formed term of this type language, under a correspondence which is preserved under the reductions in both languages and maintains the equivalence relation on their respective normal forms, by appropriately inserting applications of $\forall_\Omega$ and A (since every untyped lambda term can be translated into a term of the lambda calculus with recursive types and given the type $\mu\alpha.\,\alpha \to \alpha$ by inserting appropriate applications of fold and unfold). Since equivalence of untyped lambda terms is undecidable, typechecking a language with the above $\forall_\Omega$ and Typerec is also undecidable.

2.3 Requirements for a Solution

Let us present the central requirements for supporting intensional analysis of quantified types in a typed intermediate language.

Consider a type-directed serializer that converts a value of an arbitrary type to external representation. We will show that at the term level, the analysis must proceed inside a quantified type. Suppose we want to pickle the closure of a function of type $\tau_1 \to \tau_2$. After type-preserving compilation, this closure may be represented as a term of an existential type similar to $\exists\alpha_{env}\!:\!\Omega.\,\alpha_{env} \times (\alpha_{env} \times \tau_1 \to \tau_2)$, where the type α_{env} of the environment is held abstract. A general pickler should process this type as any other existential, and analyze its body; thus it will have to analyze the witness type for α_{env}. Even if the pair-and-code part is hard-coded as a special case, the pickler must inspect the witness type in order to pickle the environment. A similar issue arises in the comparison of two values of an existential type.

In a type-preserving compiler every phase transforms terms as well as their types to maintain type-correctness. The type transformations are defined inductively on the structure of types. For example, closure conversion would transform types as follows:

$$\begin{aligned}|\mathsf{int}| &= \mathsf{int} \\ |\tau_1 \times \tau_2| &= |\tau_1| \times |\tau_2| \\ |\tau_1 \to \tau_2| &= \exists\alpha\!:\!\Omega.\,\alpha \times (\alpha \times |\tau_1| \to |\tau_2|) \\ &\cdots\end{aligned}$$

Such type transformations [Harper and Lillibridge 1993] are conventionally expressed in a metalanguage. However, when transforming polymorphic types like $\forall\alpha\!:\!\Omega.\,\tau$, it is not obvious in general how to transform α (and other normal forms with free variables). A metalanguage transformation must define $|\alpha|$ as some type in the target language. Since α can be instantiated (at the point of type application) to any type τ', it is "too early" to choose some type constructor for $|\alpha|$. The only reasonable alternative seems to be to define $|\alpha|$ as another

variable β, appropriately introduced; in turn this implies that the transformation has been shifted to the type arguments τ'. This is not always possible, because the transformation may depend on the context of α in τ; worse yet, in a language with type analysis α may be analyzed in τ, and it would be impossible to invert the transformation of $|\tau'|$ so the results of its analysis are consistent with the source.

One way out is to use intensional type analysis to specify the transformation within the language itself, which gives the additional advantage that proving type correctness of the transformation reduces to checking well-formedness. Of course this is only possible if the analysis is defined on quantified types. A further requirement is that $|\alpha|$ must be defined as a normal form, so that the transformation can "continue operating" appropriately on the arguments at type applications, which are left unchanged.

Another serious problem in analyzing quantified types involves both the type-level and the term-level operators. Recent work on typed compilation of ML and Java [Shao 1998; 1999; League et al. 1999] has shown how to compile both languages using higher-order type constructors with arbitrarily complex kinds; there have been so far no results on type-preserving compilation of these languages which uses a fixed set of kinds. Consequently, typed intermediate languages such as FLINT [Shao 1997b] and TIL [Tarditi 1996] are based on calculi derived from F_ω [Girard 1972; Reynolds 1974], in which the quantified type variables are not restricted to a base kind Ω and can have arbitrary kinds. In the case of Java [League et al. 1999], existential quantification over higher kinds appears in the types of objects, which are prime candidates for intensional type analysis for the support of reflection. To do anything nontrivial when analyzing a package of type $\exists\alpha:\kappa.\,\tau$ at the term level, we must open the package, for which we need to know the kind κ. Having an infinite number of branches in the typecase so we can handle all possible kinds is impractical. The alternative to restrict type analysis to a finite set of kinds would make it impossible to use the known type-preserving compilation schemes for ML and Java.

Furthermore, by generalization of the result of Section 2.2 it can be shown that, if the representation of quantified types is based on higher-order abstract syntax, when the kind of the bound variable is a known constant in the corresponding branch of the Typerec construct, decidability of type-checking is lost.

This leads us to the following set of requirements for the intensional type analysis. First, the analysis must be primitively-recursive, in the style of Harper and Morrisett, the expressiveness of which has been established. Second, the analysis must proceed inside the body of a quantified type, as opposed to mapping all quantified types to the same result, for example. Third, a Typerec term analyzing a type variable must be a normal form. Fourth, the kind of quantified variables in analyzable types should not be restricted, because this would prevent the use of the current compilation techniques for higher-order typed languages. As further illustrated in Section 3.1, many interesting type-directed operations require these properties.

2.4 Problems with the Use of deBruijn Notation

The key problem in analyzing quantified types such as the polymorphic type $\forall\alpha:\Omega.\,\alpha\to\alpha$ is to determine what happens when the iteration reaches a free occurrence of the bound type variable α, or more generally a normal form which does not have a (saturated) application of a constructor of Ω in its head.

Crary and Weirich [1999] propose the use of deBruijn indices (i.e., natural numbers) to represent quantifier-bound variables. To analyze quantified types, the iterator carries an environment that maps indices to types. When the iterator reaches a type variable, which is now represented as just another constructed type (encoding a natural number), it returns the corresponding type from the environment. This method, however, has several major problems:

—The analysis is restricted to types with quantification only over variables of kind Ω. Extending it to handle a larger set of kinds is difficult, since one would have to maintain a kind environment to ensure well-formedness.

—The technique is "limited to *parametrically* polymorphic functions, and cannot account for functions that perform intensional type analysis" [Crary and Weirich 1999, Section 4.1]. For example polymorphic and existential types such as $\forall\alpha:\Omega.\,\mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots$ are not analyzable in their framework.

—A Typerec term analyzing a quantifier-bound type variable (rather, its deBruijn index) is not in normal form, hence this technique cannot be used to encode type transformations associated with closure conversion, etc.

—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). This does not break the type soundness, but opens the door for programmer mistakes.

2.5 Our Solution

To account for non-parametrically polymorphic functions, we must analyze the quantified type variable. Moreover, we want to have confluence in the type language, so β-reduction should be transparent to the iterator. This is possible only if no reduction rules apply at the head of $(\mathsf{Typerec}\ \tau\ \mathsf{of}\ \ldots)$ when τ is not (a saturated application of) a constructor of Ω. Thus the analysis "gets suspended" when it reaches a type variable of kind Ω (or an irreducible application, etc.), and resumes when the variable is substituted with a constructed type. For example, the result of analyzing the body $\alpha\to\mathsf{int}$ of the polymorphic type $\forall\alpha:\Omega.\,\alpha\to\mathsf{int}$ is

$$\mathsf{Typerec}\ (\alpha\to\mathsf{int})\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall})\ \rightsquigarrow\ \tau_{\to}\,\alpha\,(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}))\ \mathsf{int}\ \tau_{\mathsf{int}}.$$

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. The details are presented Section 3.

It is important to note that our language provides no facilities for kind analysis, thus every type function of polymorphic kind is parametrically polymor-

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

Fig. 2. Syntax of the λ_i^{ω} language

$$
\begin{array}{rcl}
\tau \to \tau' & \equiv & ((\twoheadrightarrow)\,\tau)\,\tau' \\
\forall\alpha{:}\kappa.\,\tau & \equiv & (\mathbf{\forall}\,[\kappa])\,(\lambda\alpha{:}\kappa.\,\tau) \\
\forall^{+}\chi.\,\tau & \equiv & \mathbf{\forall}^{+}(\Lambda\chi.\,\tau)
\end{array}
$$

Fig. 3. Syntactic sugar for λ_i^{ω} types

phic. Analyzing the kind κ of the bound variable α in the type $\forall\alpha{:}\kappa.\,\tau$ would let us, for instance, synthesize a type of the same kind, for every kind κ. This type could then be used to create non-terminating reduction sequences [Harper and Mitchell 1999].

3. ANALYZING QUANTIFIED TYPES

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

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

However, having an infinite number of $\forall_\kappa$ constructors is not a real option; more importantly, all of them have kinds with negative occurrences of Ω in their domains. We can replace all of them by a single constructor $\mathbf{\forall}$ of polymorphic kind $\forall\chi.\,(\chi \to \Omega) \to \Omega$ (where χ stands for a kind variable) and then instantiating it to a specific kind before forming polymorphic types. Thus our intensional polymorphic lambda calculus λ_i^{ω} (with syntax shown in Figure 2) extends F_ω with polymorphic kinds $\forall\chi.\,\kappa$ and adds the type constructor $\mathbf{\forall}$ to the type language. The polymorphic type $\forall\alpha{:}\kappa.\,\tau$ is now a derived form (Figure 3) represented as $\mathbf{\forall}\,[\kappa]\,(\lambda\alpha : \kappa.\,\tau)$; the construct $\tau\,[\kappa]$ denotes kind application at the type level.

When analyzing a type τ (of kind Ω) with the Typerec operator, the arguments of the outermost type constructor of τ must be passed to the corresponding branch of Typerec. In the case of polymorphic types represented using $\mathbf{\forall}$ these arguments are types with bound variables of arbitrary kinds. Thus the corresponding branch of the operator must bind the kind of the quantified type

variable to a kind variable; for that purpose the language provides kind abstraction ($\Lambda\chi.\,\tau$) at the type level.

Similarly, when analyzing a polymorphic type at the term level, the construct $\mathsf{typecase}$ must bind the kind of the quantified type variable to a kind variable, which necessitates the introduction of kind abstraction ($\Lambda^{+}\chi.\,e$) and kind application ($e\,[\kappa]^{+}$) at the term level. A term-level kind abstraction must be given a kind-polymorphic type, so we need a type construct $\forall^{+}\chi.\,\tau$ that binds the kind variable χ in the type τ. However our goal is to ensure that all types, now including kind-polymorphic types, can be analyzed. As with polymorphic types, the solution is to represent the type $\forall^{+}\chi.\,\tau$ as the application of a type constructor $\mathsf{\forall}^{+}$ of kind $(\forall\chi.\,\Omega) \rightarrow \Omega$ to a (type-level) kind abstraction $\Lambda\chi.\,\tau$. Thus the kinds of the constructors for types of kind Ω are as follows.

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

The kind Ω is not in a negative positions in the kind of any of these constructors' *arguments*, hence Ω is now defined inductively by these constructors. $\mathsf{Typerec}$ is then the iterator over this kind. To save space in figures we use desugared syntax for $\mathsf{Typerec}$ and $\mathsf{typecase}$, with their branches listed in fixed order and without pattern matching for their parameters; however we use friendlier syntax in examples.

The static semantics of λ_i^{ω} is displayed in Figures 4 and 5 as a set of rules for judgments, where the kind environment $\mathcal{E}$ is a list of kind variables.

Perhaps the easiest way to understand the semantics of $\mathsf{Typerec}$ is to consider first its reduction rules, given in Figure 5. Depending on the head constructor of the type τ being analyzed, $\mathsf{Typerec}$ chooses one of the branches. Similarly to Harper/Morrisett's construct, when τ is int, the result is the τ_{int} branch, and when τ is the function type $\tau_1 \rightarrow \tau_2$, the result is obtained by applying the $\tau_{\rightarrow}$ branch to the components τ_1 and τ_2 and to the results of the iteration over them.

When analyzing a polymorphic type, the reduction rule is

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

Thus the $\forall$-branch of $\mathsf{Typerec}$ 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.

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\mathcal{E};\Delta \vdash \Gamma}{\mathcal{E};\Delta;\Gamma \vdash i : \mathsf{int}} \qquad \frac{\mathcal{E} \vdash \Delta \quad \mathcal{E};\Delta,\alpha{:}\kappa;\Gamma \vdash e : \tau}{\mathcal{E};\Delta;\Gamma \vdash \Lambda\alpha{:}\kappa.\,e : \forall\alpha{:}\kappa.\,\tau} \qquad \frac{\mathcal{E};\Delta;\Gamma \vdash e : \forall\,[\kappa]\,\tau \quad \mathcal{E};\Delta \vdash \tau' : \kappa}{\mathcal{E};\Delta;\Gamma \vdash e\,[\tau'] : \tau\,\tau'}$$

$$\frac{\mathcal{E};\Delta \vdash \Gamma \quad x{:}\tau \ \mathbf{in}\ \Gamma}{\mathcal{E};\Delta;\Gamma \vdash x : \tau} \qquad \frac{\mathcal{E};\Delta;\Gamma,x{:}\tau \vdash e : \tau'}{\mathcal{E};\Delta;\Gamma \vdash \lambda x{:}\tau.\,e : \tau \to \tau'} \qquad \frac{\mathcal{E};\Delta;\Gamma \vdash e : \tau' \to \tau \quad \mathcal{E};\Delta;\Gamma \vdash e' : \tau'}{\mathcal{E};\Delta;\Gamma \vdash e\,e' : \tau}$$

$$\frac{\mathcal{E};\Delta;\Gamma,x{:}\tau \vdash v : \tau}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{fix}\,x{:}\tau.\,v : \tau}$$

where

$$\tau = \forall^{+}\chi_1 \ldots \chi_n.\ \forall\alpha_1{:}\kappa_1 \ldots \alpha_m{:}\kappa_m.\ \tau_1 \to \tau_2$$

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

Fig. 4. Formation rules of λ_i^{ω}

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

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

$$\frac{\mathcal{E};\Delta \vdash \tau : \kappa \to \kappa' \quad \alpha \notin \mathit{ftv}(\tau)}{\mathcal{E};\Delta \vdash \lambda\alpha{:}\kappa.\,\tau\,\alpha \mapsto \tau : \kappa \to \kappa'} \qquad \frac{\mathcal{E};\Delta \vdash \tau : \forall\chi'.\,\kappa \quad \chi \notin \mathit{fkv}(\tau)}{\mathcal{E};\Delta \vdash \Lambda\chi.\,\tau\,[\chi] \mapsto \tau : \forall\chi'.\,\kappa}$$

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

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

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

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

Fig. 5. Selected λ_i^ω type reduction rules

The reduction rule for analyzing a kind-polymorphic type is

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

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.

The formation rule for Typerec then follows naturally from the requirement that the above reductions preserve well-formedness. The general correspondence between the kind of a constructor of Ω and the kind of its Typerec branch [Pfenning and Paulin-Mohring 1989] is in essence that for each Ω in (a positive position in) the kinds of the arguments of the constructor we get a pair of types, one of kind Ω (the subterm itself) and the other of the kind κ of the result of the iterative invocation of Typerec. However, since λ_i^ω has no pairs at the type level, we use currying; we also have to propagate kind quantification accordingly.

Proofs of the following properties of the type language of λ_i^ω, which entail decidability of its type checking by reduction of types to their unique normal forms, can be found in Appendix A.

PROPOSITION 3.1 (STRONG NORMALIZATION). *Reduction of well-formed λ_i^ω types is strongly normalizing.*

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

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

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

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

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

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

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

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

$$\frac{\varepsilon; \varepsilon \vdash \tau' \mapsto^{*} \nu' : \Omega \qquad \nu' \text{ is a normal form}}{\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow \mathsf{typecase}[\tau]\ \nu'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}})}$$

Fig. 6. Operational semantics of λ_i^{ω}

PROPOSITION 3.2 (CONFLUENCE). *Reduction of well-formed λ_i^{ω} types is confluent.*

At the term level type analysis is carried out by the typecase construct; we do not define it as an iterator since the term language already has a recursion primitive, fix. Figure 6 displays the operational semantics of the term language of λ_i^{ω}, which shows that the $\forall$ branch of typecase receives the kind and the type abstraction carried by the type constructor $\forall$, while the $\forall^{+}$ branch gets the kind abstraction carried by $\forall^{+}$. The static semantics guarantees type safety of λ_i^{ω} programs, as shown in Appendix A.

PROPOSITION 3.3 (TYPE SAFETY). *If $\vdash e : \tau$, then either e is a value or there exists an e' such that $\vdash e' : \tau$ and $e \rightsquigarrow e'$.*

3.1 Applications

The power of intensional type analysis is in its ability to break the abstraction barriers raised by parametric polymorphism. As a consequence, however, like many other programming language features intensional type analysis "cuts both ways"—many useful properties of programs are lost in a language that offers it in its plain form. Nevertheless we believe its use is appropriate at certain levels of an implementation of a programming language, which need to know about data representation held abstract at higher levels. Typical examples include memory management, serialization, and reflection; however the detailed development of such examples is beyond the scope of this paper. In this section, we illustrate the usefulness of type-level and term-level analyses of types. We encode a type-safe marshalling primitive, and show how type classes can be simulated. The interested reader may refer to Monnier et al. [2001] for a more realistic example that involves type-checking a copying garbage collector.

To make the examples slightly more readable we will use ML-style pattern-matching syntax when writing types defined by Typerec. Instead of

$$\begin{aligned} f = \lambda\alpha{:}\Omega.\ &\mathsf{Typerec}[\kappa]\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \\ \textbf{where}\quad \tau_{\rightarrow} &= \lambda\alpha_1{:}\Omega.\ \lambda\alpha_1'{:}\kappa.\ \lambda\alpha_2{:}\Omega.\ \lambda\alpha_2'{:}\kappa.\ \tau_{\rightarrow}' \\ \tau_{\forall} &= \Lambda\chi.\ \lambda\alpha{:}\chi\rightarrow\Omega.\ \lambda\alpha'{:}\chi\rightarrow\kappa.\ \tau_{\forall}' \\ \tau_{\forall^+} &= \lambda\alpha{:}(\forall\chi.\ \Omega).\ \lambda\alpha'{:}(\forall\chi.\ \kappa).\ \tau_{\forall^+}' \end{aligned}$$

we will write

$$\begin{aligned} f\,(\mathsf{int}) &= \tau_{\mathsf{int}} \\ f\,(\alpha_1\rightarrow\alpha_2) &= \tau_{\rightarrow}'\{f\,(\alpha_1), f\,(\alpha_2)/\alpha_1', \alpha_2'\} \\ f\,(\forall\,[\chi]\,\alpha) &= \tau_{\forall}'\{\lambda\alpha_1{:}\chi.\ f\,(\alpha\,\alpha_1)/\alpha'\} \\ f\,(\forall^{+}\alpha) &= \tau_{\forall^+}'\{\Lambda\chi.\ f\,(\alpha\,[\chi])/\alpha'\}. \end{aligned}$$

3.1.1 *Marshalling.* One of the examples that Harper and Morrisett [1995] use to illustrate the power of intensional type analysis is based on the extension of ML for distributed computing proposed by Ohori and Kato [1993]. 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 type language prevents it 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^{ω} 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 Harper and Morrisett to λ_i^{ω}, we introduce a type constructor $\mathsf{Id}{:}\Omega\rightarrow\Omega$. A value of type τ has a global identifier of type $\mathsf{Id}\,\tau$. The Typerec and typecase operators are similarly extended, for example, the following rule is added to the definition of type reduction.

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

The type of the remote representation of values of type τ is $\mathsf{Tran}\,\tau$, defined by Harper and Morrisett 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^{ω} as follows.

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

To clarify the connection with the Typerec-based representation, we write the right-hand sides exactly as obtained by expanding the pattern-matching syntax introduced earlier; the redexes ostensibly present here do not exist in Typerec notation. The last clause is due to the global identifiers being marshalled as themselves.

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

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

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

$$\begin{aligned}
\mathsf{newkpid} &: \forall\alpha{:}(\forall\chi.\,\Omega).\,(\forall^{+}\chi.\,\mathsf{Tran}\,(\alpha\,[\chi]))\to\mathsf{Tran}\,(\forall^{+}\alpha)\\
\mathsf{rkapp} &: \forall\alpha{:}(\forall\chi.\,\Omega).\,\mathsf{Tran}\,(\forall^{+}\alpha)\to\forall^{+}\chi.\,\mathsf{Tran}\,(\alpha\,[\chi])
\end{aligned}$$

Operationally, given a function or a polymorphic value respectively, the new-id functions generate a new, globally unique identifier, and tell the name server to associate that identifier with the value 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 function exists; the argument is sent to this machine, and the result of the application 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 as follows (using pattern-matching syntax and implicit recursion instead of typecase and fix).

$$\begin{aligned}
\mathsf{M}\,[\mathsf{int}] &= \lambda x{:}\mathsf{int}.\,x\\
\mathsf{M}\,[\alpha_1\to\alpha_2] &= \lambda x{:}\alpha_1\to\alpha_2.\,\mathsf{newid}\,[\alpha_1]\,[\alpha_2]\,(\lambda x'{:}\mathsf{Tran}\,\alpha_1.\,\mathsf{M}\,[\alpha_2]\,(x\,(\mathsf{U}\,[\alpha_1]\,x')))\\
\mathsf{M}\,[\forall\,[\chi]\,\alpha] &= \lambda x{:}\forall\,[\chi]\,\alpha.\,\mathsf{newpid}\,[\chi]^{+}\,[\alpha]\,(\Lambda\alpha'{:}\chi.\,\mathsf{M}\,[\alpha\,\alpha']\,(x\,[\alpha']))\\
\mathsf{M}\,[\forall^{+}\alpha] &= \lambda x{:}\forall^{+}\alpha.\,\mathsf{newkpid}\,[\alpha]\,(\Lambda^{+}\chi.\,\mathsf{M}\,[\alpha\,[\chi]]\,(x\,[\chi]^{+}))\\
\mathsf{M}\,[\mathsf{Id}\,\alpha] &= \lambda x{:}\mathsf{Id}\,\alpha.\,x\\[4pt]
\mathsf{U}\,[\mathsf{int}] &= \lambda x{:}\mathsf{Tran}\,(\mathsf{int}).\,x\\
\mathsf{U}\,[\alpha_1\to\alpha_2] &= \lambda x{:}\mathsf{Tran}\,(\alpha_1\to\alpha_2).\,\lambda x'{:}\alpha_1.\,\mathsf{U}\,[\alpha_2]\,(\mathsf{rapp}\,[\alpha_1]\,[\alpha_2]\,x\,(\mathsf{M}\,[\alpha_1]\,x'))\\
\mathsf{U}\,[\forall\,[\chi]\,\alpha] &= \lambda x{:}\mathsf{Tran}\,(\forall\,[\chi]\,\alpha).\,\Lambda\alpha'{:}\chi.\,\mathsf{U}\,[\alpha\,\alpha']\,(\mathsf{rtapp}\,[\chi]^{+}\,[\alpha]\,x\,[\alpha'])\\
\mathsf{U}\,[\forall^{+}\alpha] &= \lambda x{:}\mathsf{Tran}\,(\forall^{+}\alpha).\,\Lambda^{+}\chi.\,\mathsf{U}\,[\alpha\,[\chi]]\,(\mathsf{rkapp}\,[\alpha]\,x\,[\chi]^{+})\\
\mathsf{U}\,[\mathsf{Id}\,\alpha] &= \lambda x{:}\mathsf{Tran}\,(\mathsf{Id}\,\alpha).\,x
\end{aligned}$$

We assume that a type or a kind does not need to be transformed in order to be transmitted; an implementation could use symbolic representation of types (including types of higher kind) to achieve this. A more realistic implementation would be based on a language with type-erasure semantics (Section 4),

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

where types of higher kind are represented by term-level abstractions, which could be marshalled using globally unique identifiers. However developing the details of such an implementation here would take us too far from our goal of illustrating how the new constructs of λ_i^ω enable the analysis of all run-time values.

3.1.2 *Polymorphic Equality.* Another illustration of how term-level analysis of quantified types can be used to gain access to representation information is provided by an example involving the comparison of values of existential type. At the type-level we will use the Typerec operator to define the class of types admitting equality comparisons. To make the example less 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 $\exists\,[\kappa]\,(\lambda\alpha{:}\kappa.\,\tau)$. The term constructs for introduction and elimination of existential types have the following formation rules.

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

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. Following Harper and Morrisett [1995], 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.

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

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.

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.

```
letrec
  heq : ∀α:Ω. ∀α′:Ω. Eq α → Eq α′ → Bool
    = Λα:Ω. Λα′:Ω.
        typecase[λγ:Ω. Eq γ → Eq α′ → Bool] α of
          Bool    ⇒ λx:Bool.
                      typecase[λγ:Ω. Eq γ → Bool] α′ of
                        Bool      ⇒ λy:Bool. primEqBool x y
                        ...       ⇒ ... false
          β1×β2 ⇒ λx:Eq β1 × Eq β2.
                      typecase[λγ:Ω. Eq γ → Bool] α′ of
                        β′1×β′2  ⇒ λy:Eq β′1 × Eq β′2.
                                      heq [β1] [β′1] (x.1) (y.1) and heq [β2] [β′2] (x.2) (y.2)
                        ...       ⇒ ... false
          ∃[χ] β  ⇒ λx:(∃β1:χ. Eq (β β1)).
                      typecase[λγ:Ω. Eq γ → Bool] α′ of
                        ∃[χ′] β′ ⇒ λy:(∃β′1:χ′. Eq (β′ β′1)).
                                      open x as ⟨β1:χ, xc:Eq (β β1)⟩ in
                                        open y as ⟨β′1:χ′, yc:Eq (β′ β′1)⟩ in
                                          heq [β β1] [β′ β′1] xc yc
                        ...       ⇒ ... false
          ...
in let eq : ∀α:Ω. Eq α → Eq α → Bool
        = Λα:Ω. λx:Eq α. λy:Eq α. heq [α] [α] x y
in ...
```

Fig. 7. Polymorphic equality in λ_i^ω

Consider the following two packages.

$$\begin{aligned} v &= \langle \alpha{:}\Omega = \mathsf{Bool},\ \mathsf{false}{:}\alpha \rangle \\ v' &= \langle \alpha{:}\Omega = \mathsf{Bool}\times\mathsf{Bool},\ \langle \mathsf{true},\ \mathsf{true}\rangle{:}\alpha \rangle \end{aligned}$$

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 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 the 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:

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

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

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

This term evaluates to true even though the type arguments are different. The reason is that heq actually compares the types of the values before hiding the respective 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 nontrivial 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^ω contexts, the above result is perfectly sensible.

3.2 Discussion

Before we move on, it is worthwhile to take another look at the λ_i^ω 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 type-checking many type-directed operations. Even when the focus is on 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 in order 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^ω is a minor extension (with primitive recursion) of the well-studied calculus F_2; we use the basic techniques developed for F_2 [Girard et al. 1989] to prove properties of our type language.

The kind polymorphism of λ_i^ω 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 [Harper and Mitchell 1999].

For analysis of quantified types at the term level we have the new construct $\Lambda^+\chi.\,e$ and the corresponding application. 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.

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 need kind abstraction to implement the translation to type-erasure semantics, described in Section 5.

4. TYPE-ERASURE SEMANTICS

In this section, we show that the language λ_i^ω is compatible with type-erasure semantics [Crary et al. 1998]. In a type-erasure framework, types used for the purpose of type analysis are represented at run time by terms; consequently type annotations have no run-time significance and can be erased before execution. From an implementor's point of view, this framework seems to simplify certain phases in a type-preserving compiler; most notably, typed closure conversion [Minamide et al. 1996]. Therefore, accounting for type erasure is an important step in propagating types through all phases of a type-preserving compiler.

4.1 Analyzable Elements at the Type Level

Following the ideas of Crary, Weirich, and Morrisett [1998], the run-time analysis of types is replaced by analysis of terms representing types (for instance $\mathsf{R}_{\mathsf{int}}$ represents int). The type parameters of a polymorphic function have their representation terms passed as additional term-level parameters of the function; correspondingly for every type parameter α there is a term parameter x_α which is to be bound to the term representing the type that α gets bound to. Since the type language must be kept independent of the term language in order to have decidable type checking, this analysis can only be performed at the term level. The term-level operator (now called $\mathsf{repcase}$) analyzes these representation terms.

For the analysis of representation terms to indeed mirror the analysis of types in λ_i^ω, it must hold that a term e representing type τ has e.g., the value $\mathsf{R}_{\mathsf{int}}$ if and only if $\tau = \mathsf{int}$. In [Crary et al. 1998] this is achieved by defining the representation terms so that e represents τ if and only if e has type $R\,\tau$, where R is a new type constructor, and ensuring that the type $R\,\tau$ is singleton, i.e., contains exactly one value.

Having solved the problem for representing types of kind Ω, Crary, Weirich, and Morrisett extend this solution to types of higher kinds. For instance, if α is a type parameter of kind $\Omega \to \Omega$, for the purpose of type analysis there must be a way to obtain a term representing $\alpha\,\tau$ for every type τ of kind Ω, given the terms representing α and τ. This implies that the representation of α must be a term which defines a function from $R\,\tau$ to $R\,(\alpha\,\tau)$; taking into account the requirement for polymorphism, the representation of α is of type $\forall\beta\!:\!\Omega.\, R\,\beta \to R\,(\alpha\,\beta)$. In the language of [Crary et al. 1998], which has no kind polymorphism, this construction generalizes (by induction on the structure of kinds) to the following definition of the type $R_\kappa\,(\tau)$ of terms representing the

type τ of kind κ:

$$R_\Omega(\tau) \equiv R\,\tau$$
$$R_{\kappa\to\kappa'}(\tau) \equiv \forall\beta\!:\!\kappa.\, R_\kappa(\beta) \to R_{\kappa'}(\tau\,\beta).$$

In the absence of kind abstraction and application, $R_\kappa(\tau)$ can be expanded statically for any κ. However in λ_i^ω there are polymorphic kinds and kind variables, and clearly a problem arises when κ is a variable χ. If τ is of kind χ, since the language does not offer kind analysis, there is no way to find the type $R_\chi(\tau)$ of the representation of τ, unless—similarly to the term-level representation of type variables—the type operator R_χ is provided as an extra type-level parameter α_χ of the kind abstraction for χ. Hence for every kind variable χ the translation of a λ_i^ω program to the type-erasure language must add a type variable α_χ which represents the type of term-level representations for types of kind χ. The type of terms representing τ is then $\alpha_\chi\,\tau$; hence the kind of α_χ must be $\chi \to \Omega$.

As we show next, however, the straightforward inclusion of these type-level parameters breaks the inductiveness of Ω in the type-erasure language.

Recall that the term translation introduces a new term parameter of type $R_\kappa(\alpha)$ for every type parameter α of kind κ. Thus a λ_i^ω term of type $\forall\alpha:\kappa.\,\tau$ will be translated to a term having a type of the form $\forall\alpha:\kappa.\,R_\kappa(\alpha) \to \cdots$. Therefore the translation must also change type annotations of term-level parameters of polymorphic type (as in $\lambda x:\forall\alpha:\kappa.\,\tau.\,\cdots$) to match the new types of the arguments. However, due to the polymorphism, it cannot be determined statically if a function will be invoked with an argument of polymorphic type, for instance the polymorphic identity combinator $\mathsf{I} \equiv \Lambda\alpha\!:\!\Omega.\,\lambda x\!:\!\alpha.\,x$ is invoked with itself as an argument in $\mathsf{I}\,[\forall\alpha\!:\!\Omega.\,\alpha\to\alpha]\,\mathsf{I}$. Note further that it is infeasible for the translation to change the structure of the type argument $\forall\alpha\!:\!\Omega.\,\alpha\to\alpha$, because it may be analyzed by the function using $\mathsf{Typerec}$.

There is a solution: the translation can apply to the type annotations *interpretation operators* which map the λ_i^ω-style type arguments to the types expected after the translation. In fact, since the types of the arguments are in general determined in type contexts unrelated to the context of the function, these operators cannot take advantage of free type variables and must be the same closed type operator, call it F. So I could be mapped to $\Lambda\alpha\!:\!\Omega.\,\lambda x\!:\!\mathsf{F}\,\alpha.\,x$.

Since the result of F depends on the structure of its argument (e.g., function types are just iterated through, while polymorphic types are transformed as shown above), it must be defined using type analysis. In the case of polymorphic types, $\mathsf{F}\,(\forall\,[\kappa]\,\tau)$ must yield $\forall\,[\kappa]\,(\lambda\alpha:\kappa.\,R_\kappa(\alpha) \to \mathsf{F}\,(\tau\,\alpha))$, for any κ. But here we have the old problem again: there is no way to construct R_κ for unknown κ.

The old solution—add a parameter providing R_χ, this time for a type-level kind abstraction on χ, as in the $\forall$ branch of $\mathsf{Typerec}$—is the only reasonable way out. However we must also ensure that there is an argument we can supply for this parameter, in particular when reducing a $\mathsf{Typerec}$ applied to a polymorphic type. The type R_κ depends on the kind κ carried by the constructor $\forall$, hence we can only have it if it was passed together with the $\forall$ as an additional argument. So the polymorphic type must have the shape $\underline{\forall}\,[\kappa]\,R_\kappa\,\tau$, where $\underline{\forall}$ is the

$$
\begin{array}{llcl}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \mathsf{T} \mid \kappa \to \kappa' \mid \chi \mid \forall\chi.\,\kappa \\
(\textit{types}) & \tau & ::= & \mathsf{int} \mid \twoheadrightarrow \mid \forall \mid \forall^{+} \mid R \mid T_{\mathsf{int}} \mid T_{\to} \mid T_{\forall} \mid T_{\forall^+} \mid T_R \\
& & \mid & \alpha \mid \Lambda\chi.\,\tau \mid \tau\,[\kappa] \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\,\tau' \mid \mathsf{Tagrec}[\kappa]\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\,\tau_{\to};\,\tau_{\forall};\,\tau_{\forall^+};\,\tau_R) \\
(\textit{values}) & v & ::= & i \mid \Lambda^{+}\chi.\,v \mid \Lambda\alpha{:}\kappa.\,e \mid \lambda x{:}\tau.\,e \mid \mathsf{fix}\,x{:}\tau.\,v \\
& & \mid & \mathsf{R}_{\mathsf{int}} \mid \mathsf{R}_{\to} \mid \mathsf{R}_{\to}\,[\tau] \mid \mathsf{R}_{\to}\,[\tau]\,v \mid \mathsf{R}_{\to}\,[\tau]\,v\,[\tau'] \mid \mathsf{R}_{\to}\,[\tau]\,v\,[\tau']\,v' \\
& & \mid & \mathsf{R}_{\forall} \mid \mathsf{R}_{\forall}\,[\kappa]^{+} \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau] \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau'] \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau']\,v \\
& & \mid & \mathsf{R}_{\forall^+} \mid \mathsf{R}_{\forall^+}\,[\tau] \mid \mathsf{R}_{\forall^+}\,[\tau]\,v \mid \mathsf{R}_R \mid \mathsf{R}_R\,[\tau] \mid \mathsf{R}_R\,[\tau]\,v \\
(\textit{terms}) & e & ::= & v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \mid \mathsf{repcase}[\tau]\;e\;\mathsf{of}\;(e_{\mathsf{int}};\,e_{\to};\,e_{\forall};\,e_{\forall^+};\,e_R)
\end{array}
$$

Fig. 8. Syntax of the λ_R^ω language

polymorphic type constructor in the type-erasure language, which must have kind $\forall\chi.\,(\chi \to \Omega) \to (\chi \to \Omega) \to \Omega$.

Thus the translation must replace kind applications of $\forall$; however $\forall$ is a first-class type in λ_i^ω, so for instance the type

$$(\lambda\alpha{:}(\forall\chi.\,(\chi \to \Omega) \to \Omega).\,\alpha\,[\Omega]\,(\lambda\beta{:}\Omega.\,\beta \to \beta))\,\forall$$

is well-formed. Consequently a compositional translation must augment all kind abstractions and applications with corresponding type abstractions and applications, and (in order to maintain kind-correctness) adjust the polymorphic kinds. Denoting the translation of κ by $|\kappa|$, we need

$$|\forall\chi.\,\kappa| \equiv \forall\chi.\,(\chi \to \Omega) \to |\kappa|.$$

One can expect the types of kind $\kappa \to \kappa'$ to be uneventfully translated to types of kind $|\kappa \to \kappa'| \equiv |\kappa| \to |\kappa'|$, and Ω to be mapped to Ω.

Consider now the constructor $\forall^{+}$, of kind $(\forall\chi.\,\Omega) \to \Omega$. The kind of its image under our hypothetical translation is $(\forall\chi.\,(\chi \to \Omega) \to \Omega) \to \Omega$, which has a negative occurrence of Ω in its domain. With a constructor of this kind, the kind Ω in the target language is not inductive.

As we just saw, for each kind variable we need the type operator generating the types of term-level representations of types of this variable kind. Note, however, that types of representations are not analyzed—they are only used in annotations, to verify that the terms represent the claimed types. Thus the result kind for the extra type argument does not have to be the kind of analyzable types.

This is the idea we apply in our intensional polymorphic lambda calculus with erasure, λ_R^ω. We define two kinds for the two different purposes that Ω is being used for in λ_i^ω: the kind of types of terms, and the kind of analyzable types. In λ_R^ω we reuse the name Ω for the former, while the analyzable types are called *tags*, and their kind is denoted by T. The kind Ω is defined as in λ_i^ω, so it is inductive; the kind T is also inductive, because in the kinds of its constructors only Ω and variables, but not T, occur in the domains' negative positions. In particular, the problematic $\forall^{+}$ is mapped to a constructor $T_{\forall^+}$ of kind $(\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}) \to \mathsf{T}$, in which the occurrence of Ω is acceptable, since $T_{\forall^+}$ is a constructor of T.

$$|\Omega| = \mathsf{T} \qquad |\kappa \to \kappa'| = |\kappa| \to |\kappa'| \qquad |\chi| = \chi \qquad |\forall\chi.\,\kappa| = \forall\chi.\,(\chi \to \Omega) \to |\kappa|$$

Fig. 9. Translation of λ_i^ω kinds to λ_R^ω kinds

$$\frac{\mathcal{E} \vdash \Delta}{\mathcal{E}; \Delta \vdash R_\Omega \equiv R \,:\, \mathsf{T} \to \Omega} \qquad \frac{\mathcal{E}; \Delta \vdash R_\kappa \equiv \tau \,:\, |\kappa| \to \Omega \qquad \mathcal{E}; \Delta \vdash R_{\kappa'} \equiv \tau' \,:\, |\kappa'| \to \Omega}{\begin{array}{l}\mathcal{E}; \Delta \vdash R_{\kappa\to\kappa'} \equiv \lambda\alpha{:}|\kappa \to \kappa'|.\,\forall\beta{:}|\kappa|.\,\tau\,\beta \to \tau'\,(\alpha\,\beta)\\ \quad :\, |\kappa \to \kappa'| \to \Omega\end{array}}$$

$$\frac{\mathcal{E}; \Delta \vdash \alpha_\chi \,:\, \chi \to \Omega}{\mathcal{E}; \Delta \vdash R_\chi \equiv \alpha_\chi \,:\, \chi \to \Omega} \qquad \frac{\mathcal{E}, \chi; \Delta, \alpha_\chi{:}\chi \to \Omega \vdash R_\kappa \equiv \tau \,:\, |\kappa| \to \Omega}{\begin{array}{l}\mathcal{E}; \Delta \vdash R_{\forall\chi.\,\kappa} \equiv \lambda\alpha{:}|\forall\chi.\,\kappa|.\,\forall^+\chi.\,\forall\alpha_\chi{:}\chi \to \Omega.\,\tau\,(\alpha\,[\chi]\,\alpha_\chi)\\ \quad :\, |\forall\chi.\,\kappa| \to \Omega\end{array}}$$

Fig. 10. Types of representations at higher kinds

The syntax of λ_R^ω is shown in Figure 8. The type calculus of λ_R^ω contains types and tags, distinguished by their kind; while types (of kind Ω) classify terms, tags (of kind T) are used for analysis. For every constructor that generates a type of kind Ω there is a corresponding constructor that generates a tag of kind T, e.g., for int we have T_{int}, and for $\twoheadrightarrow$ we have $T_\to$. The type analysis construct at the type level is Tagrec and it operates on tags.

At the term level we have representations for tags, since they are the analyzable elements. The primitive tags have corresponding term-level representations; for example, T_{int} is represented by $\mathsf{R}_{\mathsf{int}}$. (All well-formed applications of the term-level representation constructors, including partial applications, are values.) The type calculus in λ_R^ω includes a unary type constructor R of kind $\mathsf{T} \to \Omega$, which is used in the types of term-level representations. Given a tag τ (of kind T), the term representation of τ is constructed inductively and has type $R\,\tau$; for example, $\mathsf{R}_{\mathsf{int}}$, representing T_{int}, has type $R\,T_{\mathsf{int}}$. Semantically, as in [Crary et al. 1998], $R\,\tau$ is interpreted as a singleton type inhabited only by (the equivalence class of) the term representation of τ.

4.2 Static and Dynamic Semantics of λ_R^ω

Before we present the formation rules for λ_R^ω types and terms, it is useful to define more precisely the types of representation terms for types of higher kinds. Since the goal is to represent λ_i^ωtypes (all analyzable in λ_i^ω), the definitions follow the structure of λ_i^ω kinds. First, in Figure 9 we have the inductively defined translation of kinds from λ_i^ω to λ_R^ω. Since the analyzable elements of λ_R^ω are of kind T, the λ_i^ω kind Ω is mapped to T. On the other hand the polymorphic kind $\forall\chi.\,\kappa$ is translated to $\forall\chi.\,(\chi \to \Omega) \to |\kappa|$, since we must add a parameter for the types of representation terms for types of kind χ, but the types of representations are not analyzed, so the parameter's kind is $\chi \to \Omega$. Next, Figure 10 defines (again by induction on λ_i^ω kinds) the type operator R_κ of kind $|\kappa| \to \Omega$, mapping types of kind $|\kappa|$ to the types of their term-level representations. Note that for every kind variable χ a corresponding type variable α_χ of kind $\chi \to \Omega$ is introduced.

The formation rules for constructors for kind Ω in λ_R^ω are as in λ_i^ω, with the additional constructor R; the rules for R and the tags are displayed in

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

$$\mathcal{E} \vdash \mathsf{T}$$

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

$$\frac{\mathcal{E} \vdash \Delta}{\begin{array}{ll}\mathcal{E};\Delta \vdash R & : \mathsf{T} \to \Omega \\ \mathcal{E};\Delta \vdash T_{\mathsf{int}} & : \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\to} & : \mathsf{T} \to \mathsf{T} \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\forall} & : \forall\chi.\,(\chi \to \Omega) \to (\chi \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\forall^+} & : (\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{R} & : \mathsf{T} \to \mathsf{T}\end{array}}$$

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

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

$$\frac{\mathcal{E};\Delta \vdash \Gamma}{\begin{array}{ll}\mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\mathsf{int}} & : R\,T_{\mathsf{int}} \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\to} & : R_{\Omega\to\Omega\to\Omega}\,(T_{\to}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\forall} & : R_{\forall\chi.\,(\chi\to\Omega)\to\Omega}\,(T_{\forall}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\forall^+} & : R_{(\forall\chi.\,\Omega)\to\Omega}\,(T_{\forall^+}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{R} & : R_{\Omega\to\Omega}\,(T_{R})\end{array}}$$

$$\frac{\begin{array}{ll}\mathcal{E};\Delta \vdash \tau : \mathsf{T} \to \Omega & \\ \mathcal{E};\Delta;\Gamma \vdash e & : R\,\tau' \\ \mathcal{E};\Delta;\Gamma \vdash e_{\mathsf{int}} & : \tau\,T_{\mathsf{int}} \\ \mathcal{E};\Delta;\Gamma \vdash e_{\to} & : \forall\alpha_1{:}\mathsf{T}.\,R\,\alpha_1 \to \forall\alpha_2{:}\mathsf{T}.\,R\,\alpha_2 \to \tau\,(T_{\to}\,\alpha_1\,\alpha_2) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\forall} & : \forall^{+}\chi.\,\forall\alpha_{\chi}{:}\chi \to \Omega. \\ & \quad \forall\alpha{:}\chi \to \mathsf{T}.\,R_{\chi\to\Omega}\,\alpha \to \tau\,(T_{\forall}\,[\chi]\,\alpha_{\chi}\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\forall^+} & : \forall\alpha{:}(\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}).\,R_{\forall\chi.\,\Omega}\,\alpha \to \tau\,(T_{\forall^+}\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{R} & : \forall\alpha{:}\mathsf{T}.\,R\,\alpha \to \tau\,(T_{R}\,\alpha)\end{array}}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R}) : \tau\,\tau'}$$

Fig. 11. Formation rules for the new constructs in λ_R^ω

Figure 11. Our intention is to translate the λ_i^ω constructors of Ω, when used for type analysis, to the constructors of T, hence the kinds of the Ω constructors are mapped by $|\cdot|$ to the kinds of the corresponding tag constructors. Thus the kind of $T_\forall$ is $|\forall\chi.\,(\chi \to \Omega) \to \Omega| = \forall\chi.\,(\chi \to \Omega) \to (\chi \to \mathsf{T}) \to \mathsf{T}$; the new argument of kind $\chi \to \Omega$ can be used by the $\forall$ branch of the tag analysis construct Tagrec to form types of representation terms for types of kind χ.

To allow analysis of all tags, Tagrec includes an additional branch for the tag constructor T_R corresponding to R.

Figure 12 shows the reduction rules for Tagrec, which are similar to the reduction rules for the source language's Typerec: given a tag, it calls itself recursively on the components of the tag and then passes the result of the recursive calls, along with the original components, to the corresponding branch. Thus the reduction rule for the function tag is

$$\begin{array}{l}\mathsf{Tagrec}[\kappa]\ (T_{\to}\,\tau\,\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \\ \quad \leadsto \tau_{\to}\,\tau\,(\mathsf{Tagrec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R})) \\ \qquad\qquad \tau'\,(\mathsf{Tagrec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R})).\end{array}$$

Similarly, the reduction for the polymorphic tag is

$$\begin{array}{l}\mathsf{Tagrec}[\kappa]\ (T_{\forall}\,[\kappa]\,\tau_{\kappa}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \\ \quad \leadsto \tau_{\forall}\,[\kappa]\,\tau_{\kappa}\,\tau\,(\lambda\alpha{:}\kappa.\,\mathsf{Tagrec}[\kappa]\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R})).\end{array}$$

Figure 11 also shows the typing rules for the term representations of constructors of T and for the repcase construct. These rules use the type operator

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

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

$$\frac{\mathcal{E};\Delta,\alpha\!:\!\kappa' \vdash \mathsf{Tagrec}[\kappa]\ (\tau_2\ \alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau'\ :\ \kappa}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ (T_{\forall}\ [\kappa']\ \tau_1\ \tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau_{\forall}\ [\kappa']\ \tau_1\ \tau_2\ (\lambda\alpha\!:\!\kappa'.\,\tau')\ :\ \kappa}$$

$$\frac{\mathcal{E},\chi;\Delta,\alpha_\chi\!:\!\chi\to\Omega \vdash \mathsf{Tagrec}[\kappa]\ (\tau\ [\chi]\ \alpha_\chi)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau'\ :\ \kappa}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ (T_{\forall^+}\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau_{\forall^+}\ \tau\ (\Lambda\chi.\,\lambda\alpha_\chi\!:\!\chi\to\Omega.\,\tau')\ :\ \kappa}$$

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

Fig. 12. Non-standard reduction rules for λ_R^ω types

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

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

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

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

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\mathsf{int}}\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R) \rightsquigarrow e_{\mathsf{int}}$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\to}\ [\tau_1]\ (v_1)\ [\tau_2]\ (v_2)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R) \rightsquigarrow e_{\to}\ [\tau_1]\ (v_1)\ [\tau_2]\ (v_2)$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\forall}\ [\kappa]^+\ [\tau_\kappa]\ [\tau']\ (v)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R) \rightsquigarrow e_{\forall}\ [\kappa]^+\ [\tau_\kappa]\ [\tau']\ (v)$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\forall^+}\ [\tau']\ (v)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R) \rightsquigarrow e_{\forall^+}\ [\tau']\ (v)$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_R\ [\tau']\ (v)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R) \rightsquigarrow e_R\ [\tau']\ (v)$$

$$\frac{e \rightsquigarrow e'}{\mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R) \rightsquigarrow \mathsf{repcase}[\tau]\ e'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R)}$$

Fig. 13. Term reduction rules of λ_R^ω

R_κ as defined in Figure 10 (to save ink we are a bit sloppy with the notation, using R_κ directly as a type instead of including its formation in the premises). The typing of repcase can be derived from its reduction rules, displayed in Figure 13. The expression being analyzed must be of type $R\,\tau'$, since repcase analyzes term representation of tags. Operationally, it examines the head of the representation, selects the corresponding branch, and passes the components of the representation to the selected branch.

$$
\begin{array}{lll}
(values) & v ::= & i \mid \lambda x.e \mid \mathsf{fix}\,x.v \mid \mathsf{R_{int}} \mid \mathsf{R}_{\to} \mid \mathsf{R}_{\to}\,1 \mid \mathsf{R}_{\to}\,1\,v \mid \mathsf{R}_{\to}\,1\,v\,1 \mid \mathsf{R}_{\to}\,1\,v\,1\,v' \\
& & \mid \mathsf{R}_{\forall} \mid \mathsf{R}_{\forall}\,1 \mid \mathsf{R}_{\forall}\,1\,1 \mid \mathsf{R}_{\forall}\,1\,1\,1 \mid \mathsf{R}_{\forall}\,1\,1\,1\,v \mid \mathsf{R}_{\forall^+} \mid \mathsf{R}_{\forall^+}\,1 \mid \mathsf{R}_{\forall^+}\,1\,v \\
& & \mid \mathsf{R}_R \mid \mathsf{R}_R\,1 \mid \mathsf{R}_R\,1\,v \\
(terms) & e ::= & v \mid x \mid e\,e' \mid \mathsf{repcase}\ e\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R)
\end{array}
$$

$$
(\lambda x\!:\!\tau.\,e)\,v \leadsto_\circ e\{v/x\} \qquad (\mathsf{fix}\,x\!:\!\tau.\,v)\,v' \leadsto_\circ (v\{\mathsf{fix}\,x\!:\!\tau.\,v/x\})\,v' \qquad \frac{e \leadsto_\circ e_1}{e\,e' \leadsto_\circ e_1\,e'} \qquad \frac{e \leadsto_\circ e_1}{v\,e \leadsto_\circ v\,e_1}
$$

$$
\mathsf{repcase}\ \mathsf{R_{int}}\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R) \leadsto_\circ e_{\mathsf{int}}
$$

$$
\mathsf{repcase}\ \mathsf{R}_{\to}\,1\,v\,1\,v'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R) \leadsto_\circ e_{\to}\,1\,v\,1\,v'
$$

$$
\mathsf{repcase}\ \mathsf{R}_{\forall}\,1\,1\,1\,v\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R) \leadsto_\circ e_{\forall}\,1\,1\,1\,v
$$

$$
\mathsf{repcase}\ \mathsf{R}_{\forall^+}\,1\,v\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R) \leadsto_\circ e_{\forall^+}\,1\,v
$$

$$
\mathsf{repcase}\ \mathsf{R}_R\,1\,v\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R) \leadsto_\circ e_R\,1\,v
$$

$$
\frac{e \leadsto_\circ e'}{\mathsf{repcase}\ e\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R) \leadsto_\circ \mathsf{repcase}\ e'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R)}
$$

Fig. 14. Syntax and semantics of the untyped language $\lambda_R^{\omega\circ}$

$$
\begin{array}{ccc}
(\Lambda^{+}\chi.\,v)^\circ = \lambda_.v^\circ & (e\,[\kappa]^{+})^\circ = e^\circ\,1 & \mathsf{R_{int}}^\circ = \mathsf{R_{int}} \\
(\Lambda\alpha\!:\!\kappa.\,e)^\circ = \lambda_.e^\circ & (e\,[\tau])^\circ = e^\circ\,1 & \mathsf{R}_{\to}{}^\circ = \mathsf{R}_{\to} \\
(\lambda x\!:\!\tau.\,e)^\circ = \lambda x.e^\circ & (e\,e')^\circ = e^\circ\,e'^\circ & \mathsf{R}_{\forall}{}^\circ = \mathsf{R}_{\forall} \\
(\mathsf{fix}\,x\!:\!\tau.\,v)^\circ = \mathsf{fix}\,x.v^\circ & x^\circ = x & \mathsf{R}_{\forall^+}{}^\circ = \mathsf{R}_{\forall^+} \\
 & i^\circ = i & \mathsf{R}_R{}^\circ = \mathsf{R}_R
\end{array}
$$

$$
(\mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^+};\, e_R))^\circ \ = \ \mathsf{repcase}\ e^\circ\ \mathsf{of}\ (e_{\mathsf{int}}{}^\circ;\, e_{\to}{}^\circ;\, e_{\forall}{}^\circ;\, e_{\forall^+}{}^\circ;\, e_R{}^\circ)
$$

Fig. 15. Translation of λ_R^{ω} to $\lambda_R^{\omega\circ}$

The language λ_R^{ω} enjoys the following properties.

PROPOSITION 4.1 (TYPE REDUCTION). *Reduction of well-formed types is strongly normalizing and confluent.*

PROPOSITION 4.2 (TYPE SAFETY). *If* $\vdash e : \tau$, *then either* e *is a value, or there exists a term* e' *such that* $e \leadsto e'$ *and* $\vdash e' : \tau$.

The proofs of these propositions are similar to the proofs of the corresponding propositions for λ_i^{ω}.

4.3 The Untyped Language

To demonstrate that the types in λ_R^{ω} are not necessary for computation, we present an untyped language $\lambda_R^{\omega\circ}$ in Figure 14, and a translation from λ_R^{ω} to $\lambda_R^{\omega\circ}$ in Figure 15; the expression 1 in these figures is the integer constant. The untyped language has the following property which shows that term reduction $\leadsto_\circ$ in it parallels the term reduction in λ_R^{ω}.

PROPOSITION 4.3. *If* $e \leadsto e_1$, *then* $e^\circ \leadsto_\circ e_1{}^\circ$.

Corollary 4.4. *If $\vdash e : \tau$ and $e^\circ \leadsto_\circ e'_0$, then there exists e' such that $\vdash e' : \tau$ and $e'^\circ = e'_0$.*

Proof From $\vdash e : \tau$ by Proposition 4.2 we have that either e is a value, or $e \leadsto e'$ for some e' such that $\vdash e' : \tau$. Since (by inspection of the definition of values in Figures 13 and 14) the erasure of a value is a value, and if v is a value, then $v \leadsto_\circ e'_0$ for no e'_0, it follows that e is not a value. Thus there exists e' such that $e \leadsto e'$ and $\vdash e' : \tau$. By Proposition 4.3, $e^\circ \leadsto_\circ e'^\circ$. By induction on the structure of untyped terms, for any untyped term e_0 at most one derivation exists deriving $e_0 \leadsto_\circ e_1$ for some e_1. Thus from $e^\circ \leadsto_\circ e'_0$ and $e^\circ \leadsto_\circ e'^\circ$ we have $e'_0 = e'^\circ$. □

Corollary 4.5 (Safety of $\lambda_R^{\omega\circ}$). *If $\vdash e : \tau$ and $e^\circ \leadsto_\circ^* e'_0$ for some untyped term e'_0, then either e'_0 is a value, or there exists an untyped term e''_0 such that $e'_0 \leadsto_\circ e''_0$.*

Proof By induction on the length of the reduction sequence deriving $e^\circ \leadsto_\circ^* e'_0$. If the length is zero, by Proposition 4.2 either e is a value, in which case its erasure e° is a value, or $e \leadsto e'$ for some e', and then by Proposition 4.3 $e^\circ \leadsto_\circ e'^\circ$. In the inductive case, assuming the statement holds for all sequences of length n and given a sequence of length $n+1$, let the first step of the sequence be $e^\circ \leadsto_\circ e'_1$. Then by Corollary 4.4 there exists e' such that $\vdash e' : \tau$ and $e'^\circ = e'_1$. Since the rest of the sequence derives $e'_1 \leadsto_\circ^* e'_0$, the result follows directly by the inductive hypothesis applied to e'. □

The translation replaces type and kind applications (abstractions) by dummy applications (abstractions), instead of erasing them. This peculiarity is due to the semantics of the fix construct in our typed languages: A type or kind application of a fixpoint term reduces by unfolding the fixpoint. The translation inserts the dummy applications and parameters to ensure the corresponding unfolding in the untyped language.

5. TRANSLATION FROM λ_i^ω TO λ_R^ω

In this section, we show a translation from λ_i^ω to λ_R^ω. The languages differ mainly in two ways. First, the type calculus in λ_R^ω is split into tags and types, with types used solely for type checking and tags used for analysis. Since any type argument in λ_i^ω can potentially be analyzed, type passing in λ_i^ω will be translated to tag passing in λ_R^ω, while type annotations will be reconstructed from the tags. Second, the typecase operator in λ_i^ω must be translated to a repcase operating on term representations of tags.

Figure 16 shows the translation of λ_i^ω types into λ_R^ω tags. The primitive type constructors are mapped to the corresponding primitive tag constructors. Notice all closed λ_i^ω types in normal forms are translated into similarly structured λ_R^ω tag types (except that $T_\forall$ now takes an extra argument)—this is important since any nontrivial structural changes may alter the results of analysis via Typerec. The Typerec is translated to a Tagrec; the translation inserts an arbitrarily chosen result of the correct kind into the branch for the T_R tag since the source contains no such branch.

The term translation is shown in Figure 17. The translation must maintain two invariants. First, for kind variable χ in scope there is a corresponding

$$|\Lambda\chi.\,\tau| = \Lambda\chi.\,\lambda\alpha_\chi : \chi \to \Omega.\,|\tau| \qquad |\lambda\alpha : \kappa.\,\tau| = \lambda\alpha : |\kappa|.\,|\tau| \qquad |\mathsf{int}| = T_{\mathsf{int}} \qquad |\forall| = T_\forall$$

$$|\tau\,[\kappa]| = |\tau|\,[|\kappa|]\,R_\kappa \qquad |\tau\,\tau'| = |\tau|\,|\tau'| \qquad |\to| = T_\to \qquad |\forall^+| = T_{\forall^+}$$

$$|\alpha| = \alpha$$

$$|\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})| = \mathsf{Tagrec}[|\kappa|]\ |\tau|\ \mathsf{of}\ (|\tau_{\mathsf{int}}|;\ |\tau_\to|;\ |\tau_\forall|;\ |\tau_{\forall^+}|;\ \lambda_ : \mathsf{T}.\,\lambda_ : |\kappa|.\,|\tau_{\mathsf{int}}|)$$

Fig. 16. Translation of λ_i^ω types to λ_R^ω tags

$$\begin{aligned}
|i| &= i\\
|x| &= x\\
|\Lambda^+\chi.\,e| &= \Lambda^+\chi.\,\Lambda\alpha_\chi : \chi \to \Omega.\,|e|\\
|e\,[\kappa]^+| &= |e|\,[|\kappa|]^+\,[R_\kappa]\\
|\Lambda\alpha : \kappa.\,e| &= \Lambda\alpha : |\kappa|.\,\lambda x_\alpha : R_\kappa\,\alpha.\,|e|\\
|e\,[\tau]| &= |e|\,[|\tau|]\,\Re(\tau)\\
|\lambda x : \tau.\,e| &= \lambda x : \mathsf{F}\,|\tau|.\,|e|\\
|e\,e'| &= |e|\,|e'|\\
|\mathsf{fix}\,x : \tau.\,v| &= \mathsf{fix}\,x : \mathsf{F}\,|\tau|.\,|v|
\end{aligned}$$

$$\begin{aligned}
&|\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_\to;\ e_\forall;\ e_{\forall^+})|\\
&\quad = \mathsf{repcase}[\lambda\alpha : \mathsf{T}.\,\mathsf{F}\,(|\tau|\,\alpha)]\ \Re(\tau')\ \mathsf{of}\\
&\qquad\quad \mathsf{R}_{\mathsf{int}} \Rightarrow |e_{\mathsf{int}}|\\
&\qquad\quad \mathsf{R}_\to \Rightarrow |e_\to|\\
&\qquad\quad \mathsf{R}_\forall \Rightarrow |e_\forall|\\
&\qquad\quad \mathsf{R}_{\forall^+} \Rightarrow |e_{\forall^+}|\\
&\qquad\quad \mathsf{R}_R \Rightarrow \Lambda\beta : \mathsf{T}.\,\mathsf{fix}\,x : R\,\beta \to \mathsf{F}\,(|\tau|\,(T_R\,\beta)).\,\lambda x' : R\,\beta.\,x\,x'
\end{aligned}$$

Fig. 17. Translation of λ_i^ω terms to λ_R^ω terms

$$\begin{aligned}
\Re(\mathsf{int}) &= \mathsf{R}_{\mathsf{int}}\\
\Re(\to) &= \Lambda\alpha : \mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\,\Lambda\beta : \mathsf{T}.\,\lambda x_\beta : R\,\beta.\,\mathsf{R}_\to\,[\alpha]\,(x_\alpha)\,[\beta]\,(x_\beta)\\
\Re(\forall) &= \Lambda^+\chi.\,\Lambda\alpha_\chi : \chi \to \Omega.\,\Lambda\alpha : \chi \to \mathsf{T}.\,\lambda x_\alpha : R_{\chi\to\Omega}\,(\alpha).\,\mathsf{R}_\forall\,[\chi]^+\,[\alpha_\chi]\,[\alpha]\,(x_\alpha)\\
\Re(\forall^+) &= \Lambda\alpha : (\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}).\,\lambda x_\alpha : R_{\forall\chi.\,\Omega}\,(\alpha).\,\mathsf{R}_{\forall^+}\,[\alpha]\,(x_\alpha)\\
\Re(\alpha) &= x_\alpha\\
\Re(\Lambda\chi.\,\tau) &= \Lambda^+\chi.\,\Lambda\alpha_\chi : \chi \to \Omega.\,\Re(\tau)\\
\Re(\tau\,[\kappa]) &= \Re(\tau)\,[|\kappa|]^+\,[R_\kappa]\\
\Re(\lambda\alpha : \kappa.\,\tau) &= \Lambda\alpha : |\kappa|.\,\lambda x_\alpha : R_\kappa\,\alpha.\,\Re(\tau)\\
\Re(\tau\,\tau') &= \Re(\tau)\,[|\tau'|]\,(\Re(\tau'))
\end{aligned}$$

$$\begin{aligned}
&\Re(\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})) =\\
&\quad (\mathsf{fix}\,\mathsf{f} : \forall\alpha : \mathsf{T}.\,R\,\alpha \to R\ (\tau^*\,\alpha).\\
&\qquad \Lambda\alpha : \mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\\
&\qquad \mathsf{repcase}[\lambda\alpha : \mathsf{T}.\,R\ (\tau^*\,\alpha)]\ x_\alpha\ \mathsf{of}\\
&\qquad\quad \mathsf{R}_{\mathsf{int}} \Rightarrow \Re(\tau_{\mathsf{int}})\\
&\qquad\quad \mathsf{R}_\to \Rightarrow \Lambda\alpha : \mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\,\Lambda\beta : \mathsf{T}.\,\lambda x_\beta : R\,\beta.\\
&\qquad\qquad\qquad \Re(\tau_\to)\,[\alpha]\,(x_\alpha)\,[\tau^*\,\alpha]\,(\mathsf{f}\,[\alpha]\,x_\alpha)\,[\beta]\,(x_\beta)\,[\tau^*\,\beta]\,(\mathsf{f}\,[\beta]\,x_\beta)\\
&\qquad\quad \mathsf{R}_\forall \Rightarrow \Lambda^+\chi.\,\Lambda\alpha_\chi : \chi \to \Omega.\,\Lambda\alpha : \chi \to \mathsf{T}.\,\lambda x_\alpha : R_{\chi\to\Omega}\,(\alpha).\\
&\qquad\qquad\qquad \Re(\tau_\forall)\,[\chi]^+\,[\alpha_\chi]\,[\alpha]\,(x_\alpha)\,[\lambda\beta : \chi.\,\tau^*\,(\alpha\,\beta)]\,(\Lambda\beta : \chi.\,\lambda x_\beta : \alpha_\chi\,\beta.\,\mathsf{f}\,[\alpha\,\beta]\,(x_\alpha\,[\beta]\,x_\beta))\\
&\qquad\quad \mathsf{R}_{\forall^+} \Rightarrow \Lambda\alpha : (\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}).\,\lambda x_\alpha : R_{\forall\chi.\,\Omega}\,(\alpha).\\
&\qquad\qquad\qquad \Re(\tau_{\forall^+})\,[\alpha]\,(x_\alpha)\,[\Lambda\chi.\,\lambda\alpha_\chi : \chi \to \Omega.\,\tau^*\,(\alpha\,[\chi]\,\alpha_\chi)]\\
&\qquad\qquad\qquad\quad (\Lambda^+\chi.\,\Lambda\alpha_\chi : \chi \to \Omega.\,\mathsf{f}\,[\alpha\,[\chi]\,\alpha_\chi]\,(x_\alpha\,[\chi]^+\,[\alpha_\chi]))\\
&\qquad\quad \mathsf{R}_R \Rightarrow \Lambda\alpha : \mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\,\Re(\tau_{\mathsf{int}}))\\
&\quad [|\tau|]\\
&\quad \Re(\tau)\\
&\quad \textbf{where}\\
&\qquad \tau^* = |\lambda\alpha : \Omega.\,\mathsf{Typerec}[\kappa]\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})|
\end{aligned}$$

Fig. 18. Representation of λ_i^ω types as λ_R^ω terms

type variable α_χ, which gives the type of the term representation for a tag of kind χ. At every kind application, the translation uses the function R_κ (Figure 10) to compute this type. (R_κ is defined at the meta-level by induction on κ, but for every κ the result is a type in λ_R^ω.) Thus, the translations of kind abstractions and kind applications introduce an additional type abstraction and application, respectively.

Second, for every type variable α in scope there is a term variable x_α, providing the term representation of α. At every type application, the translation uses the meta-function $\Re$ (Figure 18) to construct this representation. Furthermore, type application is replaced by a type application to the tag corresponding to the type argument, followed by an application to the term representation of this tag.

Programs in λ_R^ω pass tags at run time since only tags can be analyzed. However, abstractions and the fixpoint construct must still carry type annotations for type checking. These annotations are reconstructed from the tags corresponding to the λ_i^ω types by the *tag interpretation operator* F, defined within the λ_R^ω type language using Tagrec. Since the annotations are always of kind Ω, this operator must map tags of kind T to types of kind Ω. In pattern-matching syntax the operator is defined as follows:

$$
\begin{aligned}
\mathsf{F}\,(T_{\mathsf{int}}) &= \mathsf{int} \\
\mathsf{F}\,(T_{\rightarrow}\,\alpha_1\,\alpha_2) &= \mathsf{F}\,(\alpha_1) \rightarrow \mathsf{F}\,(\alpha_2) \\
\mathsf{F}\,(T_{\forall}\,[\chi]\,\alpha_\chi\,\alpha) &= \forall\beta:\chi.\,\alpha_\chi\,\beta \rightarrow \mathsf{F}\,(\alpha\,\beta) \\
\mathsf{F}\,(T_{\forall^+}\,\alpha) &= \forall^{+}\chi.\,\forall\alpha_\chi:\chi\rightarrow\Omega.\,\mathsf{F}\,(\alpha\,[\chi]\,\alpha_\chi) \\
\mathsf{F}\,(T_{R}\,\alpha) &= R\,\alpha
\end{aligned}
$$

The function F maps a tag representing a λ_i^ω type to the corresponding λ_R^ω type. Thus it maps the tag T_{int} to the type int, and recursively converts the components of other tags to the corresponding types before combining the results with Ω constructors. The branch for the T_R tag is irrelevant, as long as it has the correct kind, since the language λ_R^ω is only intended as a target for translation from λ_i^ω—the only interesting programs in λ_R^ω are the ones translated from λ_i^ω, in which the T_R branch of F is never reached.

The tag interpretation function F is another example of a type transformation defined within the type language instead of at the meta level (cf. the discussion in Section 2.3).

The following two properties hold since the branches of F have no free type or kind variables.

LEMMA 5.1. $(\mathsf{F}\,(\tau))\{\tau'/\alpha\} = \mathsf{F}\,(\tau\{\tau'/\alpha\})$

LEMMA 5.2. $(\mathsf{F}\,(\tau))\{\kappa/\chi\} = \mathsf{F}\,(\tau\{\kappa/\chi\})$

We show the term representation of types in Figure 18. The primitive type constructors get translated to the corresponding term representation. The representations of type and kind functions are similar to the term translation of type and kind abstractions. The only involved case is the term representation of a $\mathsf{Typerec}$. Since $\mathsf{Typerec}$ is recursive, we use a combination of a $\mathsf{repcase}$ and a fix. Note that the translation of type-level kind polymorphism in λ_i^ω requires

term-level kind polymorphism in λ_R^ω, e.g., the $\forall$ branch of Typerec is translated using term-level kind abstraction.

By induction on the structure of kinds we have the following properties of the translation.

LEMMA 5.3. $|\kappa\{\kappa'/\chi\}| = |\kappa|\{|\kappa'|/\chi\}$

LEMMA 5.4. $(R_\kappa)\{|\kappa'|, R_{\kappa'}/\chi', \alpha_{\chi'}\} = R_{\kappa\{\kappa'/\chi'\}}$

In the following propositions the original λ_i^ω kind environment Δ is extended with a kind environment $\Delta(\mathcal{E})$ which binds a type variable α_χ of kind $\chi \to \Omega$ for each $\chi \in \mathcal{E}$, under the assumption that $\alpha_\chi \notin \Delta$. Similarly the term-level translations extend the type environment Γ with $\Gamma(\Delta)$, binding a variable x_α of type $R_\kappa\, \alpha$ for each type variable α bound in Δ with kind κ.

PROPOSITION 5.5 (WELL-FORMEDNESS OF TRANSLATED TYPES). *If* $\mathcal{E}; \Delta \vdash \tau : \kappa$ *holds in* λ_i^ω, *then* $|\mathcal{E}|; |\Delta|, \Delta(\mathcal{E}) \vdash |\tau| : |\kappa|$ *holds in* λ_R^ω.

Proof Follows directly by induction over the structure of τ. □

PROPOSITION 5.6 (TYPES OF REPRESENTATION TERMS). *If* $\mathcal{E}; \Delta \vdash \tau : \kappa$ *and* $\mathcal{E}; \Delta \vdash \Gamma$ *hold in* λ_i^ω, *then* $|\mathcal{E}|; |\Delta|, \Delta(\mathcal{E}); |\Gamma|, \Gamma(\Delta) \vdash \Re(\tau) : R_\kappa\, |\tau|$ *holds in* λ_R^ω.

Proof By induction over the structure of τ. The only interesting case is that of a kind application which uses Lemma 5.4. □

PROPOSITION 5.7 (WELL-FORMEDNESS OF TRANSLATED TERMS). *If* $\mathcal{E}; \Delta; \Gamma \vdash e : \tau$ *holds in* λ_i^ω, *then* $|\mathcal{E}|; |\Delta|, \Delta(\mathcal{E}); |\Gamma|, \Gamma(\Delta) \vdash |e| : \mathsf{F}\, |\tau|$ *holds in* λ_R^ω.

Proof By induction over the structure of e, using Lemmas 5.1 and 5.2. □

6. RELATED WORK

The work of Harper and Morrisett [1995] 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 and Weirich [1999] 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" [Crary and Weirich 1999, 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 [1998] proposes another framework for intensional type analysis; however, he allows the analysis of types only at the term level but not at the type level. Yang [1998] presents some approaches to enable type-safe programming of type-indexed values in ML which is similar to term-level analysis of types. Having term-level analysis only is not enough for applications such as type safe garbage collectors [Monnier et al. 2001] (where type-level analysis is used to certify the memory interface between the mutator and the collector).

Necula [1998] proposed the ideas of a certifying compiler and implemented a certifying compiler for a type-safe subset of C. Morrisett et al. [1998] 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 by Pierce et al. [1989]. Pfenning and Paulin-Mohring [1989] show how inductively defined types can be represented by closed types. They also construct representations of all primitive recursive functions over inductively defined types.

Despeyroux et al. [1997] proposed a technique for performing primitive recursion on higher-order abstract syntax in a logic framework. While there are some similarities on the surface, there are also many subtle differences between their systems and ours. In their system, there is a clear distinction between the object language (the logic they are representing) and the meta language (the underlying logic framework) so that the adequacy (for the representation) can be established. Our system, however, are not trying to representing one language inside another; instead, our calculus is just a typed intermediate language. Despeyroux et al. use modal logic to clearly identify the set of terms that can be analyzed, while we use kind polymorphism to achieve parametricity. Their method does not apply in our context because it can only analyze fully closed terms. Our technique, on the other hand, does support intensional analysis on types with free variables.

The type erasure semantics follows the idea proposed in Crary et al. [1998]. However, they consider a language that analyzes only first order types. Extending the analysis to arbitrary types makes the translation into a type erasure semantics much more complicated. The splitting of the type calculus into types and tags and defining an interpretation function to map between the two are related to the ideas proposed by Crary and Weirich for the language LX [Crary and Weirich 1999].

The erasure framework also resembles the dictionary passing style in Haskell [Peterson and Jones 1993]. The term representation of a type may be viewed as the dictionary corresponding to the type. However, the authors consider dictionary passing in an untyped calculus; moreover, they do not consider the intensional analysis of types. Dubois et al. [1995] also pass explicit type representations in their extensional polymorphism scheme. However, they do not provide a mechanism for connecting a type to its representation. Minamide's type-lifting procedure [Minamide 1997] is also related to our work. His procedure maintains interrelated constraints between type parameters; however, his language does not support intensional type analysis. Weirich [2000] pre-

sented a technique for encoding intensional analysis of (non-quantified) types using Haskell type classes, but her scheme only supports term-level analysis.

7. CONCLUSIONS

We presented a type-theoretic framework for analyzing quantified (such as polymorphic and existential) types. It makes possible the analysis of arbitrary quantified types both at the type level and at the term level. The central idea is to use higher-order abstract syntax to represent quantified types, and to introduce parametric kind polymorphism to retain inductiveness of the analyzable kind. The analysis is not restricted to parametric quantified type; it can also handle types that analyze the quantified type variable. The calculus λ_i^ω is sound and its type checking remains decidable.

We also gave a translation of our calculus to a language λ_R^ω with type-erasure semantics, which is more suitable for implementation due to the elimination of run-time significance of types; the latter point is made clear by establishing a correspondence with the reductions in an untyped language.

For completeness of the type analysis and for the purpose of this translation both λ_i^ω and λ_R^ω introduce kind abstraction and application at the term level, and a corresponding type constructor $\forall^+$. This does not increase the complexity of the type languages, which are essentially F_2 with primitive recursion. The term languages become extensions of Girard's λU calculus [Girard 1972], hence not strongly normalizing; however strong normalization is not a requirement for a term-level language, and our term languages already includes the general recursion construct fix, necessary in a realistic programming language.

APPENDIX

A. PROPERTIES OF λ_i^ω

A.1 Soundness of the λ_i^ω Type System

The rules for single-step reduction in λ_i^ω are shown in Figure 6, and are standard except for those involving the typecase construct. The typecase chooses a branch depending on the head constructor of the type being analyzed and passes to it as arguments the subterms of the type. For example, while analyzing the polymorphic type $\forall\,[\kappa]\,\tau$, it applies the $\forall$ branch ($e_\forall$ in the figure) to the kind κ and the type function τ. The last rule ensures that the type being analyzed is first reduced to its unique normal form (Theorem A.48).

We prove soundness of the system using contextual semantics in the style of Wright and Felleisen [1994]. The reduction rules for the redexes r are as shown in Figure 6, and we define evaluation contexts E in Figure 19. We assume unique variable names. The notation $\vdash e : \tau$ is used a shorthand for $\varepsilon;\varepsilon;\varepsilon \vdash e : \tau$.

Since the reduction of typecase in λ_i^ω depends on the form of a type, we introduce normal forms ν of types in Figure 20.

LEMMA A.1. *If* $\varepsilon;\varepsilon \vdash \nu : \Omega$, *then* ν *is one of* int, $\nu_1 \to \nu_2$, $\forall\,[\kappa]\,\nu_1$, *or* $\forall^+\nu_1$.

Proof Since ν is well-formed in an empty environment, it does not contain any free type or kind variables. Therefore ν can not be a ν^0 since the head of

$$
\begin{array}{lll}
(value) & v & ::= i \mid \Lambda^{+}\chi.\, e \mid \Lambda\alpha{:}\kappa.\, e \mid \lambda x{:}\tau.\, e \mid \mathsf{fix}\, x{:}\tau.\, v \\[1em]
(context) & E & ::= [\,] \mid E\, e \mid v\, E \mid E\,[\tau] \mid E\,[\kappa]^{+} \\[1em]
(redex) & r & ::= (\Lambda^{+}\chi.\, e)\,[\kappa]^{+} \mid (\Lambda\alpha{:}\kappa.\, e)\,[\tau] \mid (\lambda x{:}\tau.\, e)\, v \\
& & \mid (\mathsf{fix}\, x{:}\tau.\, v)\,[\kappa]^{+} \mid (\mathsf{fix}\, x{:}\tau.\, v)\,[\tau'] \mid (\mathsf{fix}\, x{:}\tau.\, v)\, v' \\
& & \mid \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \\
& & \mid \mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \mid \mathsf{typecase}[\tau]\ (\tau \rightarrow \tau')\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \\
& & \mid \mathsf{typecase}[\tau]\ (\forall\,[\kappa]\,\tau)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \mid \mathsf{typecase}[\tau]\ (\forall^{+}\tau)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}})
\end{array}
$$

Fig. 19. Term contexts

$$
\begin{array}{lll}
\nu^0 & ::= & \alpha \mid \nu^0\,\nu \mid \nu^0\,[\kappa] \mid \mathsf{Typerec}[\kappa]\ \nu^0\ \mathsf{of}\ (\nu_{\mathsf{int}};\, \nu_{\rightarrow};\, \nu_{\forall};\, \nu_{\forall^{+}}) \\
\nu & ::= & \nu^0 \mid \mathsf{int} \mid \twoheadrightarrow \mid (\twoheadrightarrow)\,\nu \mid (\twoheadrightarrow)\,\nu\,\nu' \mid \forall \mid \forall\,[\kappa] \mid \forall\,[\kappa]\,\nu \mid \forall^{+} \mid \forall^{+}\nu \\
& \mid & \lambda\alpha{:}\kappa.\,\nu,\ \text{where}\ \forall\nu^0.\ \nu \neq \nu^0\,\alpha\ \text{or}\ \alpha \in \mathit{ftv}(\nu^0) \\
& \mid & \Lambda\chi.\,\nu,\ \text{where}\ \forall\nu^0.\ \nu \neq \nu^0\,[\chi]\ \text{or}\ \chi \in \mathit{fkv}(\nu^0)
\end{array}
$$

Fig. 20. Normal forms in the λ_i^{ω} type language

a ν^0 is a type variable. The lemma now follows by inspecting the remaining possibilities for ν. □

LEMMA A.2 (DECOMPOSITION OF TERMS). *If $\vdash e:\tau$, then either e is a value or it can be decomposed into unique E and r such that $e = E\{r\}$.*

This is proved by induction over the derivation of $\vdash e:\tau$, using Lemma A.1 in the case of the typecase construct.

COROLLARY A.3 (PROGRESS). *If $\vdash e:\tau$, then either e is a value or there exists an e' such that $e \mapsto e'$.*

Proof By Lemma A.2, we know that if $\vdash e:\tau$ and e is not a value, then there exist some E and redex e_1 such that $e = E\{e_1\}$. Since e_1 is a redex, there exists a contraction e_2 such that $e_1 \leadsto e_2$. Therefore $e \mapsto e'$ for $e' = E\{e_2\}$. □

LEMMA A.4. *If $\vdash E\{e\}:\tau$, then there exists a τ' such that $\vdash e:\tau'$, and for all e' such that $\vdash e':\tau'$ we have $\vdash E\{e'\}:\tau$.*

Proof The proof is by induction on the derivation of $\vdash E\{e\}:\tau$. The different forms of E are handled similarly; we will show only one case here.

—**case** $E = E_1\, e_1$: We have that $\vdash (E_1\{e\})\, e_1 : \tau$. By the typing rules, this implies that $\vdash E_1\{e\}:\tau_1 \rightarrow \tau$, for some τ_1. By induction, there exists a τ' such that $\vdash e:\tau'$ and for all e' such that $\vdash e':\tau'$, we have that $\vdash E_1\{e'\}:\tau_1 \rightarrow \tau$. Therefore $\vdash (E_1\{e'\})\, e_1:\tau$. □

As usual, the proof of soundness depends on several substitution lemmas; these are shown below. The proofs are fairly straightforward and proceed by induction on the derivation of the judgments. The notion of substitution is extended to environments in the usual way.

LEMMA A.5. *If $\mathcal{E}, \chi \vdash \kappa$ and $\mathcal{E} \vdash \kappa'$, then $\mathcal{E} \vdash \kappa\{\kappa'/\chi\}$.*

LEMMA A.6. *If* $\mathcal{E},\chi;\Delta \vdash \tau : \kappa$ *and* $\mathcal{E} \vdash \kappa'$, *then* $\mathcal{E};\Delta\{\kappa'/\chi\} \vdash \tau\{\kappa'/\chi\} : \kappa\{\kappa'/\chi\}$.

LEMMA A.7. *If* $\mathcal{E},\chi;\Delta;\Gamma \vdash e : \tau$ *and* $\mathcal{E} \vdash \kappa$, *then* $\mathcal{E};\Delta\{\kappa/\chi\};\Gamma\{\kappa/\chi\} \vdash e\{\kappa/\chi\} : \tau\{\kappa/\chi\}$.

LEMMA A.8. *If* $\mathcal{E};\Delta,\alpha{:}\kappa' \vdash \tau : \kappa$ *and* $\mathcal{E};\Delta \vdash \tau' : \kappa'$, *then* $\mathcal{E};\Delta \vdash \tau\{\tau'/\alpha\} : \kappa$.

LEMMA A.9. *If* $\mathcal{E};\Delta,\alpha:\kappa;\Gamma \vdash e : \tau$ *and* $\mathcal{E};\Delta \vdash \tau' : \kappa$, *then* $\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e\{\tau'/\alpha\} : \tau\{\tau'/\alpha\}$.

Proof We prove this by induction on the structure of e. We demonstrate the proof here only for a few cases; the rest follow analogously.

—**case** $e = e_1\,[\tau_1]$: We have that $\mathcal{E};\Delta \vdash \tau' : \kappa$. and also that $\mathcal{E};\Delta,\alpha:\kappa;\Gamma \vdash e_1\,[\tau_1] : \tau$. By the typing rule for a type application we get that $\mathcal{E};\Delta,\alpha{:}\kappa;\Gamma \vdash e_1 : \forall\beta:\kappa_1.\,\tau_2$, $\mathcal{E};\Delta,\alpha:\kappa \vdash \tau_1 : \kappa_1$, and $\tau = \tau_2\{\tau_1/\beta\}$. By induction on e_1, $\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_1\{\tau'/\alpha\} : \forall\beta{:}\kappa_1.\,\tau_2\{\tau'/\alpha\}$. By Lemma A.8, $\mathcal{E};\Delta \vdash \tau_1\{\tau'/\alpha\} : \kappa_1$. Therefore

$$\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash (e_1\{\tau'/\alpha\})\,[\tau_1\{\tau'/\alpha\}] : (\tau_2\{\tau'/\alpha\})\{\tau_1\{\tau'/\alpha\}/\beta\}.$$

But this is equivalent to

$$\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash (e_1\{\tau'/\alpha\})\,[\tau_1\{\tau'/\alpha\}] : (\tau_2\{\tau_1/\beta\})\{\tau'/\alpha\}.$$

—**case** $e = e_1\,[\kappa_1]^{+}$: We have that $\mathcal{E};\Delta,\alpha{:}\kappa;\Gamma \vdash e_1\,[\kappa_1]^{+} : \tau$ and $\mathcal{E};\Delta \vdash \tau' : \kappa$. By the typing rule for kind application, $\mathcal{E};\Delta,\alpha{:}\kappa;\Gamma \vdash e_1 : \forall\chi.\,\tau_1$, $\tau = \tau_1\{\kappa_1/\chi\}$, and $\mathcal{E} \vdash \kappa_1$. By induction on e_1, $\mathcal{E};\Delta;\Gamma \vdash e_1\{\tau'/\alpha\} : \forall\chi.\,\tau_1\{\tau'/\alpha\}$. Therefore $\mathcal{E};\Delta;\Gamma \vdash (e_1\{\tau'/\alpha\})\,[\kappa_1]^{+} : (\tau_1\{\tau'/\alpha\})\{\kappa_1/\chi\}$. Since χ does not occur free in τ' we have $(\tau_1\{\tau'/\alpha\})\{\kappa_1/\chi\} = (\tau_1\{\kappa_1/\chi\})\{\tau'/\alpha\}$.

—**case** $e = \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^+})$: We have that $\mathcal{E};\Delta \vdash \tau' : \kappa$ and $\mathcal{E};\Delta,\alpha{:}\kappa;\Gamma \vdash \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^+}) : \tau_0\,\tau_1$. Using Lemma A.8 on the kind derivation of τ_0 and τ_1, and the inductive assumption on the typing rules for the subterms we get

$$\begin{array}{l}
\mathcal{E};\Delta \vdash \tau_0\{\tau'/\alpha\} : \Omega \rightarrow \Omega\\
\mathcal{E};\Delta \vdash \tau_1\{\tau'/\alpha\} : \Omega\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\mathsf{int}}\{\tau'/\alpha\} : (\tau_0\ \mathsf{int})\{\tau'/\alpha\}\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\rightarrow}\{\tau'/\alpha\} : (\forall\alpha_1{:}\Omega.\,\forall\alpha_2{:}\Omega.\,\tau_0\,(\alpha_1 \rightarrow \alpha_2))\{\tau'/\alpha\}\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\forall}\{\tau'/\alpha\} : (\forall^{+}\chi.\,\forall\alpha{:}\chi \rightarrow \Omega.\,\tau_0\,(\forall\,[\chi]\,\alpha))\{\tau'/\alpha\}\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\forall^+}\{\tau'/\alpha\} : (\forall\alpha{:}\forall\chi.\,\Omega.\,\tau_0\,(\forall^{+}\alpha))\{\tau'/\alpha\}
\end{array}$$

The above typing judgments are equivalent to

$$\begin{array}{l}
\mathcal{E};\Delta \vdash \tau_0\{\tau'/\alpha\} : \Omega \rightarrow \Omega\\
\mathcal{E};\Delta \vdash \tau_1\{\tau'/\alpha\} : \Omega\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\mathsf{int}}\{\tau'/\alpha\} : (\tau_0\{\tau'/\alpha\})\ \mathsf{int}\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\rightarrow}\{\tau'/\alpha\} : \forall\alpha_1{:}\Omega.\,\forall\alpha_2{:}\Omega.\,(\tau_0\{\tau'/\alpha\})\,(\alpha_1 \rightarrow \alpha_2)\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\forall}\{\tau'/\alpha\} : \forall^{+}\chi.\,\forall\alpha{:}\chi \rightarrow \Omega.\,(\tau_0\{\tau'/\alpha\})\,(\forall\,[\chi]\,\alpha)\\
\mathcal{E};\Delta;\Gamma\{\tau'/\alpha\} \vdash e_{\forall^+}\{\tau'/\alpha\} : \forall\alpha{:}\forall\chi.\,\Omega.\,(\tau_0\{\tau'/\alpha\})\,(\forall^{+}\alpha)
\end{array}$$

from which the statement of the lemma follows directly. □

LEMMA A.10. *If* $\mathcal{E};\Delta;\Gamma,x:\tau' \vdash e : \tau$ *and* $\mathcal{E};\Delta;\Gamma \vdash e' : \tau'$*, then* $\mathcal{E};\Delta;\Gamma \vdash e\{e'/x\} : \tau$.

Proof Proved by induction over the structure of e. The different cases are proved similarly. We will show only two cases here.

—**case** $e = \Lambda\alpha:\kappa.\,e_1$: We have that $\mathcal{E};\Delta;\Gamma,x:\tau' \vdash \Lambda\alpha:\kappa.\,e_1 : \forall\alpha:\kappa.\tau$ and $\mathcal{E};\Delta;\Gamma \vdash e' : \tau'$. Since e can always be alpha-converted, we assume that α is not previously defined in Δ. This implies $\mathcal{E};\Delta,\alpha:\kappa;\Gamma,x:\tau' \vdash e_1 : \tau$. Since α is not free in e', we have $\mathcal{E};\Delta,\alpha:\kappa;\Gamma \vdash e' : \tau'$. By induction, $\mathcal{E};\Delta,\alpha:\kappa;\Gamma \vdash e_1\{e'/x\} : \tau$. Hence $\mathcal{E};\Delta;\Gamma \vdash \Lambda\alpha:\kappa.\,e_1\{e'/x\} : \forall\alpha:\kappa.\,\tau$.

—**case** $e = \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^+})$: We have that $\mathcal{E};\Delta;\Gamma \vdash e' : \tau'$ and $\mathcal{E};\Delta;\Gamma,x:\tau' \vdash \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^+}) : \tau_0\,\tau_1$. By the typecase typing rule we get

$$
\begin{array}{l}
\mathcal{E};\Delta \vdash \tau_0 : \Omega \rightarrow \Omega \text{ and}\\
\mathcal{E};\Delta \vdash \tau_1 : \Omega \text{ and}\\
\mathcal{E};\Delta;\Gamma,x:\tau' \vdash e_{\mathsf{int}} : \tau_0\,\mathsf{int} \text{ and}\\
\mathcal{E};\Delta;\Gamma,x:\tau' \vdash e_{\rightarrow} : \forall\alpha_1:\Omega.\,\forall\alpha_2:\Omega.\,\tau_0\,(\alpha_1 \rightarrow \alpha_2) \text{ and}\\
\mathcal{E};\Delta;\Gamma,x:\tau' \vdash e_{\forall} : \forall^+\chi.\,\forall\alpha:\chi \rightarrow \Omega.\,\tau_0\,(\forall\,[\chi]\,\alpha) \text{ and}\\
\mathcal{E};\Delta;\Gamma,x:\tau' \vdash e_{\forall^+} : \forall\alpha:\forall\chi.\,\Omega.\,\tau_0\,(\forall^+\alpha)
\end{array}
$$

Applying the inductive hypothesis to each of the subterms $e_{\mathsf{int}}, e_{\rightarrow}, e_{\forall}, e_{\forall^+}$ yields directly the claim. □

DEFINITION A.11. e evaluates to e' (written $e \mapsto e'$) if there exist E, e_1, and e_2 such that $e = E\{e_1\}$ and $e' = E\{e_2\}$ and $e_1 \leadsto e_2$.

THEOREM A.12 (SUBJECT REDUCTION). *If* $\vdash e:\tau$ *and* $e \mapsto e'$*, then* $\vdash e':\tau$.

Proof By Lemma A.2, e can be decomposed into unique E and unique redex e_1 such that $e = E\{e_1\}$. By definition, $e' = E\{e_2\}$ and $e_1 \leadsto e_2$. By Lemma A.4, there exists a τ' such that $\vdash e_1:\tau'$. By the same lemma, all we need to prove is that $\vdash e_2:\tau'$ holds. This is proved by considering each possible redex in turn. We will show only two cases, the rest follow similarly.

—**case** $e_1 = (\mathsf{fix}\,x:\tau_1.\,v)\,v'$: Then $e_2 = (v\{\mathsf{fix}\,x:\tau_1.\,v/x\})\,v'$. We have that $\vdash (\mathsf{fix}\,x:\tau_1.\,v)\,v' : \tau'$. By the typing rules for term application we get that for some τ_2, $\vdash \mathsf{fix}\,x:\tau_1.\,v : \tau_2 \rightarrow \tau'$ and $\vdash v' : \tau_2$. By the typing rule for fix we get that, $\vdash \tau_1 = \tau_2 \rightarrow \tau'$ and $\varepsilon;\varepsilon;\varepsilon,x:\tau_2 \rightarrow \tau' \vdash v : \tau_2 \rightarrow \tau'$. Using Lemma A.10 and the typing rule for application, we obtain the desired judgment $\vdash (v\{\mathsf{fix}\,x:\tau_1.\,v/x\})\,v':\tau'$.

—**case** $e_1 = \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^+})$: If τ_1 is not in normal form, the reduction is to $e_2 = \mathsf{typecase}[\tau_0]\ \nu_1\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^+})$, where $\varepsilon;\varepsilon \vdash \tau_1 \mapsto^* \nu_1 : \Omega$. The latter implies $\varepsilon;\varepsilon \vdash \tau_0\,\tau_1 = \tau_0\,\nu_1 : \Omega$, hence $\vdash e_2 : \tau'$ follows directly from $\vdash e_1:\tau'$.
If τ_1 is in normal form ν_1, by the second premise of the typing rule for typecase and Lemma A.1 we have four cases for ν_1. In each case the contraction has the desired type $\tau_0\,\nu_1$, according to the corresponding premises of the typecase typing rule and the rules for type and kind applications. □

(β_1)	$(\lambda\alpha:\kappa.\,\tau)\,\tau'$	$\leadsto$	$\tau\{\tau'/\alpha\}$	
(β_2)	$(\Lambda\chi.\,\tau)\,[\kappa]$	$\leadsto$	$\tau\{\kappa/\chi\}$	
(η_1)	$\lambda\alpha:\kappa.\,(\tau\,\alpha)$	$\leadsto$	τ	$\alpha \notin \mathit{ftv}(\tau)$
(η_2)	$\Lambda\chi.\,(\tau\,[\chi])$	$\leadsto$	τ	$\chi \notin \mathit{fkv}(\tau)$
(t_1)	$\mathsf{Typerec}[\kappa]\,\mathsf{int}\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+})$	$\leadsto$	τ_{int}	
(t_2)	$\mathsf{Typerec}[\kappa]\,(\tau\to\tau')\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+})$	$\leadsto$	$\tau_{\to}\,\tau\,(\mathsf{Typerec}[\kappa]\,\tau\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+}))$ $\tau'\,(\mathsf{Typerec}[\kappa]\,\tau'\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+}))$	
(t_3)	$\mathsf{Typerec}[\kappa]\,(\forall\,[\kappa]\,\tau)\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+})$	$\leadsto$	$\tau_{\forall}\,[\kappa]\,\tau\,(\lambda\alpha:\kappa.\mathsf{Typerec}[\kappa]\,(\tau\,\alpha)\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+}))$	
(t_4)	$\mathsf{Typerec}[\kappa]\,(\forall^+\tau)\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+})$	$\leadsto$	$\tau_{\forall^+}\,\tau\,(\Lambda\chi.\,\mathsf{Typerec}[\kappa]\,(\tau\,[\chi])\,\mathsf{of}\,(\tau_{\mathsf{int}};\tau_{\to};\tau_{\forall};\tau_{\forall^+}))$	

Fig. 21. Type reductions

A.2 Strong Normalization in the λ_i^ω Type Language

Notation. In this section we occasionally write $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\vec{\tau})$ instead of $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$. We use $\overline{A}$ to denote a sequence $\{A_1,\ldots,A_n\}$, and $B\{\overline{A}/\overline{a}\}$ for the result of applying a sequence of substitutions.

The single-step reduction relation $\leadsto$ on types is the union of the relations defined by the rules in Figure 21.

LEMMA A.13. *If* $\tau_1 \leadsto \tau_2$, *then* $\tau_1\{\tau/\alpha\} \leadsto \tau_2\{\tau/\alpha\}$.

Proof Consider the possible reductions from τ_1 to τ_2.
case β_1: In this case, $\tau_1 = (\lambda\beta:\kappa.\,\tau')\,\tau''$ and $\tau_2 = \tau'\{\tau''/\beta\}$, for some τ',τ'', and β, and without loss of generality β can be assumed not to occur free in τ. This implies that

$$\tau_1\{\tau/\alpha\} = (\lambda\beta:\kappa.\,(\tau'\{\tau/\alpha\}))\,(\tau''\{\tau/\alpha\})$$

The right-hand side reduces by β_1 to $(\tau'\{\tau/\alpha\})\{\tau''\{\tau/\alpha\}/\beta\}$. Since β does not occur free in τ, this type is equivalent to $(\tau'\{\tau''/\beta\})\{\tau/\alpha\}$.
case β_2: In this case, $\tau_1 = (\Lambda\chi.\,\tau')\,[\kappa]$ and $\tau_2 = \tau'\{\kappa/\chi\}$. Hence $\tau_1\{\tau/\alpha\} = (\Lambda\chi.\,\tau'\{\tau/\alpha\})\,[\kappa]$, which reduces by β_2 to $(\tau'\{\tau/\alpha\})\{\kappa/\chi\} = (\tau'\{\kappa/\chi\})\{\tau/\alpha\}$.
case η_1: We have that $\tau_1 = \lambda\beta:\kappa.\,(\tau'\,\beta)$, $\tau_2 = \tau'$, and β does not occur free in τ' and τ. Hence $\tau_1\{\tau/\alpha\} = \lambda\beta:\kappa.\,((\tau'\{\tau/\alpha\})\,\beta)$. Since β still does not occur free in $\tau'\{\tau/\alpha\}$, this type reduces by η_1 to $\tau'\{\tau/\alpha\}$.
case η_2: In this case, $\tau_1 = \Lambda\chi.\,\tau'\,[\chi]$, $\tau_2 = \tau'$, and χ does not occur free in τ' and τ. We get that $\tau_1\{\tau/\alpha\} = \Lambda\chi.\,(\tau'\{\tau/\alpha\})\,[\chi]$. Since χ does not occur free in $\tau'\{\tau/\alpha\}$, by η_2 this type reduces to $\tau'\{\tau/\alpha\}$.
The cases of reductions of $\mathsf{Typerec}$ are similar; we show only
case t_3: $\tau_1 = \mathsf{Typerec}[\kappa]\ (\forall\,[\kappa']\,\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$ and

$$\tau_2 = \tau_{\forall}\,[\kappa']\,\tau'\,(\lambda\beta:\kappa'.\,\mathsf{Typerec}[\kappa]\ (\tau'\,\beta)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))$$

We get that

$$\tau_1\{\tau/\alpha\} = \mathsf{Typerec}[\kappa]\ (\forall\,[\kappa']\,\tau'\{\tau/\alpha\})\ \mathsf{of}\ (\tau_{\mathsf{int}}\{\tau/\alpha\};\ \tau_{\to}\{\tau/\alpha\};\ \tau_{\forall}\{\tau/\alpha\};\ \tau_{\forall^+}\{\tau/\alpha\})$$

This reduces by t_3 to

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

which is syntactically equivalent to $\tau_2\{\tau/\alpha\}$. □

LEMMA A.14. *If* $\tau_1 \rightsquigarrow \tau_2$*, then* $\tau_1\{\kappa'/\chi'\} \rightsquigarrow \tau_2\{\kappa'/\chi'\}$*.*

Proof By case analysis of the type reduction relation.
case β_1: In this case, $\tau_1 = (\lambda\beta\!:\!\kappa.\,\tau')\,\tau''$ and $\tau_2 = \tau'\{\tau''/\beta\}$. This implies that

$$\tau_1\{\kappa'/\chi'\} = (\lambda\beta\!:\!\kappa\{\kappa'/\chi'\}.\,\tau'\{\kappa'/\chi'\})\,\tau''\{\kappa'/\chi'\},$$

which reduces by β_1 to $(\tau'\{\kappa'/\chi'\})\{\tau''\{\kappa'/\chi'\}/\beta\}$, which in turn is equivalent to $(\tau'\{\tau''/\beta\})\{\kappa'/\chi'\}$.
case β_2: In this case, $\tau_1 = (\Lambda\chi.\,\tau')\,[\kappa]$ and $\tau_2 = \tau'\{\kappa/\chi\}$. Then

$$\tau_1\{\kappa'/\chi'\} = (\Lambda\chi.\,\tau'\{\kappa'/\chi'\})\,[\kappa\{\kappa'/\chi'\}],$$

which reduces by β_2 to $\tau'\{\kappa'/\chi'\}\{\kappa\{\kappa'/\chi'\}/\chi\}$. Since w.l.o.g. χ is not free in κ', the latter is equivalent to $(\tau'\{\kappa/\chi\})\{\kappa'/\chi'\}$.
The other cases follow similarly. □

DEFINITION A.15. A type τ is *strongly normalizable* if every reduction sequence from τ terminates into a normal form (with no redexes). We use $\nu(\tau)$ to denote the length of the largest reduction sequence from τ to a normal form.

DEFINITION A.16. We define *neutral types*, n, as
$n_0 \;::=\; \Lambda\chi.\,\tau \mid \lambda\alpha\!:\!\kappa.\,\tau$
$n \;::=\; \alpha \mid n_0\,\tau \mid n\,\tau \mid n_0\,[\kappa] \mid n\,[\kappa] \mid \mathsf{Typerec}[\kappa]\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^+})$

DEFINITION A.17. A *reducibility candidate* (also referred to as simply a *candidate*) of kind κ is a set $\mathcal{C}$ of types of kind κ such that

(1) if $\tau \in \mathcal{C}$, then τ is strongly normalizable.
(2) if $\tau \in \mathcal{C}$ and $\tau \rightsquigarrow \tau'$, then $\tau' \in \mathcal{C}$.
(3) if τ is neutral and if for all τ' such that $\tau \rightsquigarrow \tau'$, we have that $\tau' \in \mathcal{C}$, then $\tau \in \mathcal{C}$.

This implies that the candidates are never empty since if α has kind κ, then α belongs to candidates of kind κ.

DEFINITION A.18. Let κ be an arbitrary kind. Let $\mathcal{C}_{\mathsf{int}}$ be a candidate of kind κ, $\mathcal{C}_{\rightarrow}$ be a candidate of kind $\Omega \rightarrow \kappa \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa$, $\mathcal{C}_{\forall}$ be a candidate of kind $\forall\chi.\,(\chi \rightarrow \Omega) \rightarrow (\chi \rightarrow \kappa) \rightarrow \kappa$, and $\mathcal{C}_{\forall^+}$ be a candidate of kind $(\forall\chi.\,\Omega) \rightarrow (\forall\chi.\,\kappa) \rightarrow \kappa$. The set R_Ω of types of kind Ω is then defined as

$$\{\tau \mid \textit{for all}\; \tau_{\mathsf{int}} \in \mathcal{C}_{\mathsf{int}}, \tau_{\rightarrow} \in \mathcal{C}_{\rightarrow}, \tau_{\forall} \in \mathcal{C}_{\forall},\; \textit{and}\; \tau_{\forall^+} \in \mathcal{C}_{\forall^+},$$
$$\mathsf{Typerec}[\kappa]\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^+}) \in \mathcal{C}_\kappa\}.$$

LEMMA A.19. R_Ω *is a candidate of kind* Ω*.*

Proof We will prove R_Ω satisfies the requirements of Definition A.17. In each of the cases below, let $\mathcal{C}_0$, $\mathcal{C}_{\rightarrow}$, $\mathcal{C}_{\forall}$, and $\mathcal{C}_{\forall^+}$ be candidates of kinds κ, $\Omega \rightarrow \kappa \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa$, $\forall\chi.\,(\chi \rightarrow \Omega) \rightarrow (\chi \rightarrow \kappa) \rightarrow \kappa$, and $(\forall\chi.\,\Omega) \rightarrow (\forall\chi.\,\kappa) \rightarrow \kappa$, respectively, and τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ be elements of the respective candidates.

(1) Suppose that τ is in R_Ω, and let $\tau_1 \equiv \mathsf{Typerec}[\kappa]\;\tau\;\mathsf{of}\;(\vec{\tau})$. By Definition A.18 τ_1 belongs to $\mathcal{C}_0$. By property 1 of Definition A.17, τ' is strongly normalizable, therefore τ is strongly normalizable.

(2) Suppose $\tau \leadsto \tau'$, and again let $\tau_1 \equiv \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\vec{\tau})$. Then we have that $\tau_1 \leadsto \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\vec{\tau})$. Since $\tau_1 \in \mathcal{C}_0$, by property 2 of Definition A.17 $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\vec{\tau})$ belongs to $\mathcal{C}_0$. Therefore, by Definition A.18, $\tau' \in R_\Omega$.

(3) Suppose τ is neutral, and for all τ', if $\tau \leadsto \tau'$, then $\tau' \in R_\Omega$. Let $\tau_1 \equiv \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\vec{\tau})$. Note that, since by assumption τ_{int}, $\tau_\rightarrow$, $\tau_\forall$, and $\tau_{\forall^+}$ are members of the appropriate candidates, by Definition A.18 this implies that $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\vec{\tau}) \in \mathcal{C}_0$. Furthermore, the four branches are strongly normalizable, hence we can proceed by induction on the length of τ_1 defined by $len(\tau_1) \equiv \nu(\tau_{\mathsf{int}}) + \nu(\tau_\rightarrow) + \nu(\tau_\forall) + \nu(\tau_{\forall^+})$ to prove that τ_1 always reduces to a type that belongs to $\mathcal{C}_0$.
   - $len(\tau_1) = 0$. Then $\tau_1 \leadsto \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\vec{\tau})$ is the only possible reduction since τ is neutral.
   - $len(\tau_1) = k+1$. In this case the inductive hypothesis is that any type of the form $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\vec{\tau})$ of length k reduces to a type that belongs to $\mathcal{C}_0$. Now τ_1 can either reduce to $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\vec{\tau})$, which (we showed) is in $\mathcal{C}_0$, or to
     $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau'_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$, when $\tau_{\mathsf{int}} \leadsto \tau'_{\mathsf{int}}$,
     $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau'_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$, when $\tau_\rightarrow \leadsto \tau'_\rightarrow$,
     $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau'_\forall;\ \tau_{\forall^+})$, when $\tau_\forall \leadsto \tau'_\forall$, or
     $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau'_{\forall^+})$, when $\tau_{\forall^+} \leadsto \tau'_{\forall^+}$.
     By property 2 of Definition A.17, each of τ'_{int}, $\tau'_\rightarrow$, $\tau'_\forall$, and $\tau'_{\forall^+}$ also belongs to the appropriate candidate, and the length of each of the reducts is k. Therefore, by the inductive hypothesis, each of the reducts belongs to $\mathcal{C}_0$.

   Therefore $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\vec{\tau})$ always reduces to a type that belongs to $\mathcal{C}_0$. By property 3 of Definition A.17, $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\vec{\tau})$ also belongs to $\mathcal{C}_0$. Hence $\tau \in R_\Omega$. □

DEFINITION A.20. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two candidates of kinds κ_1 and κ_2. We then define the set $\mathcal{C}_1 \rightarrow \mathcal{C}_2$, of types of kind $\kappa_1 \rightarrow \kappa_2$, as $\{\tau \mid \text{ for all } \tau_1 \in \mathcal{C}_1,\ \tau\,\tau_1 \in \mathcal{C}_2\}$.

LEMMA A.21. *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are candidates of kinds κ_1 and κ_2, then $\mathcal{C}_1 \rightarrow \mathcal{C}_2$ is a candidate of kind $\kappa_1 \rightarrow \kappa_2$.*

Proof We will prove $\mathcal{C}_1 \rightarrow \mathcal{C}_2$ satisfies the requirements of Definition A.17.

(1) Suppose τ of kind $\kappa_1 \rightarrow \kappa_2$ is in $\mathcal{C}_1 \rightarrow \mathcal{C}_2$. By definition, if $\tau' \in \mathcal{C}_1$, then $\tau\,\tau' \in \mathcal{C}_2$. Since $\mathcal{C}_2$ is a candidate, $\tau\,\tau'$ is strongly normalizable. Therefore, τ must be strongly normalizable since for every sequence of reductions $\tau \leadsto \tau_1 \ldots \tau_k \ldots$, there is a corresponding sequence of reductions $\tau\,\tau' \leadsto \tau_1\,\tau' \ldots \tau_k\,\tau' \ldots$.

(2) Suppose τ of kind $\kappa_1 \rightarrow \kappa_2$ belongs to $\mathcal{C}_1 \rightarrow \mathcal{C}_2$, and $\tau \leadsto \tau'$. Let $\tau_1 \in \mathcal{C}_1$; then by Definition A.20 $\tau\,\tau_1 \in \mathcal{C}_2$. But $\tau\,\tau_1 \leadsto \tau'\,\tau_1$. By Definition A.17, property 2, $\tau'\,\tau_1 \in \mathcal{C}_2$; therefore, $\tau' \in \mathcal{C}_1 \rightarrow \mathcal{C}_2$.

(3) Consider a neutral τ of kind $\kappa_1 \rightarrow \kappa_2$. Suppose that for all τ', if $\tau \leadsto \tau'$, then $\tau' \in \mathcal{C}_1 \rightarrow \mathcal{C}_2$. Consider $\tau\,\tau_1$ where $\tau_1 \in \mathcal{C}_1$. Since τ_1 is strongly normalizable, we use induction over $\nu(\tau_1)$. If $\nu(\tau_1) = 0$, then $\tau\,\tau_1 \leadsto \tau'\,\tau_1$. But $\tau'\,\tau_1 \in \mathcal{C}_2$ (by the assumption on τ'), and since τ is neutral, no other reduction is

possible. If $\nu(\tau_1) > 0$, then $\tau_1 \leadsto \tau_1'$ for some τ_1'. In this case, $\tau\,\tau_1$ may reduce to either $\tau'\,\tau_1$ or to $\tau\,\tau_1'$. We saw that the first reduct is in $\mathcal{C}_2$. By property 2 of Definition A.17, $\tau_1' \in \mathcal{C}_1$; also $\nu(\tau_1') < \nu(\tau_1)$. By the inductive hypothesis we get that $\tau\,\tau_1' \in \mathcal{C}_2$. Then by property 3 of Definition A.17, $\tau\,\tau_1 \in \mathcal{C}_2$. This implies that $\tau \in \mathcal{C}_1 \to \mathcal{C}_2$. □

DEFINITION A.22. Let $\overline{\chi}$ be the sequence of all free kind variables in kind κ, $\overline{\kappa}$ be a sequence of closed kinds of the same length, and $\overline{\mathcal{C}}$ be a sequence of candidates of the corresponding kinds. Define the set $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ of types of kind $\kappa\{\overline{\kappa}/\overline{\chi}\}$ inductively on the structure of κ as follows:

—if $\kappa = \Omega$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}] = R_\Omega$.
—if $\kappa = \chi_i$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}] = \mathcal{C}_i$.
—if $\kappa = \kappa_1 \to \kappa_2$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}] = \mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}] \to \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$.
—if $\kappa = \forall\chi.\,\kappa'$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ is the set of types τ of kind $\kappa\{\overline{\kappa}/\overline{\chi}\}$ such that for every kind κ'' and candidate $\mathcal{C}''$ of kind κ'', $\tau\,[\kappa''] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}''/\overline{\chi}, \chi]$.

LEMMA A.23. *$\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ is a reducibility candidate of kind $\kappa\{\overline{\kappa}/\overline{\chi}\}$.*

Proof We prove the statement by induction on the structure of κ. For $\kappa = \Omega$, the result follows from Lemma A.19; for $\kappa = \chi$, directly from the definition of $\mathcal{S}_\chi[\overline{\mathcal{C}}/\overline{\chi}]$. If $\kappa = \kappa_1 \to \kappa_2$, we can apply the inductive hypothesis on κ_1 and κ_2 and Lemma A.21. We only need to prove the case for $\kappa = \forall\chi.\,\kappa'$. Let $\overline{\chi}$ containing all the free kind variables of κ.

(1) Suppose $\tau \in \mathcal{S}_{\forall\chi.\,\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$. By Definition A.22, for any kind κ_1 and corresponding candidate $\mathcal{C}_1$, $\tau\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$. Applying the inductive hypothesis to κ', we get that $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$ is a candidate. Therefore, $\tau\,[\kappa_1]$ is strongly normalizable, which implies that τ is strongly normalizable.
(2) Suppose $\tau \in \mathcal{S}_{\forall\chi.\,\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$ and $\tau \leadsto \tau_1$. For any kind κ_1 and corresponding candidate $\mathcal{C}_1$, by definition, $\tau\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$. But $\tau\,[\kappa_1] \leadsto \tau_1\,[\kappa_1]$. By the inductive hypothesis on κ' we have that $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$ is a candidate; then by property 2 of Definition A.17, $\tau_1\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$. Therefore, $\tau_1 \in \mathcal{S}_{\forall\chi.\,\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$.
(3) Consider a neutral τ so that for all τ_1, if $\tau \leadsto \tau_1$, then $\tau_1 \in \mathcal{S}_{\forall\chi.\,\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$. Consider an arbitrary kind κ_1 and a corresponding candidate $\mathcal{C}_1$. Since τ is neutral, the only possible reduction of $\tau\,[\kappa_1]$ is to $\tau_1\,[\kappa_1]$. By the assumption on τ_1 we have $\tau_1\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$. By the inductive hypothesis on κ' it follows that $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$ is a candidate. By property 3 of Definition A.17, $\tau\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$. Therefore $\tau \in \mathcal{S}_{\forall\chi.\,\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$. □

LEMMA A.24. $\mathcal{S}_{\kappa\{\kappa'/\chi'\}}[\overline{\mathcal{C}}/\overline{\chi}] = \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]/\overline{\chi}, \chi']$

Proof The proof is by induction over the structure of κ. We will show only the case for polymorphic kinds, the others follow directly by induction. Suppose $\kappa = \forall\chi''.\,\kappa''$. Then the LHS is the set of types τ of kind $(\forall\chi''.\,\kappa''\{\kappa'/\chi'\})\{\overline{\kappa}/\overline{\chi}\}$ such that for every kind κ''' and corresponding candidate $\mathcal{C}'''$, $\tau\,[\kappa''']$ belongs to $\mathcal{S}_{\kappa''\{\kappa'/\chi'\}}[\overline{\mathcal{C}}, \mathcal{C}'''/\overline{\chi}, \chi'']$. Applying the inductive hypothesis to κ'', this is equal to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}''', \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'''/\overline{\chi}, \chi'']/\overline{\chi}, \chi'', \chi']$. But χ'' does not occur free in κ' (variables in κ' can always be renamed), hence $\tau\,[\kappa''']$ is in $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}''', \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]/\overline{\chi}, \chi'', \chi']$.

The RHS consists of types τ' of kind $(\forall\chi''.\,\kappa'')\{\overline{\kappa}, \kappa'\{\overline{\kappa}/\overline{\chi}\}/\overline{\chi}, \chi'\}$ (which is equivalent to $(\forall\chi''.\,\kappa''\{\kappa'/\chi'\})\{\overline{\kappa}/\overline{\chi}\}$) such that for every kind κ''' and corresponding candidate $\mathcal{C}'''$, $\tau'\,[\kappa''']$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{\chi}], \mathcal{C}'''/\overline{\chi}, \chi', \chi'']$, i.e., the same set as the LHS. □

DEFINITION A.25. From Lemma A.23, we know that for every kind κ and sequences of variables $\overline{\chi}$ and candidates $\overline{\mathcal{C}}$, $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ is a candidate of kind $\kappa\{\overline{\kappa}/\overline{\chi}\}$, that $\mathcal{S}_{\Omega\to\kappa\to\Omega\to\kappa\to\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ is a candidate of kind $(\Omega \to \kappa \to \Omega \to \kappa \to \kappa)\{\overline{\kappa}/\overline{\chi}\}$, that $\mathcal{S}_{\forall\chi.\,(\chi\to\Omega)\to(\chi\to\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ is a candidate of kind $(\forall\chi.\,(\chi \to \Omega) \to (\chi \to \kappa) \to \kappa)\{\overline{\kappa}/\overline{\chi}\}$, and $\mathcal{S}_{(\forall\chi.\,\Omega)\to(\forall\chi.\,\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ is a candidate of kind $((\forall\chi.\,\Omega) \to (\forall\chi.\,\kappa) \to \kappa)\{\overline{\kappa}/\overline{\chi}\}$. Throughout the rest of the section, leaving $\kappa, \overline{\chi}$, and $\overline{\mathcal{C}}$ to be determined by the context, we define $\vec{\tau}$ as a quadruple of types τ_{int}, $\tau_{\to}$, $\tau_{\forall}$, and $\tau_{\forall^+}$, which are elements of the above respective candidates.

LEMMA A.26. $\mathsf{int} \in R_\Omega = \mathcal{S}_\Omega[\overline{\mathcal{C}}/\overline{\chi}]$.

Proof Suffices to prove that $\tau \equiv \mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\vec{\tau})$ is in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\kappa}]$, whenever $\vec{\tau}$ are constrained by Definition A.25. The proof is by induction on $len(\tau) \equiv \nu(\tau_{\mathsf{int}}) + \nu(\tau_{\to}) + \nu(\tau_{\forall}) + \nu(\tau_{\forall^+})$.

—$len(\tau) = 0$. Then τ can reduce only to τ_{int}, which is by Definition A.25 in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\kappa}]$.

—$len(\tau) = k + 1$. Then the inductive hypothesis is that any Typerec of length k on int reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. By property 3 of Definition A.17 this implies that any Typerec of length k on int belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. When $len(\tau) = k+1$, we have that τ either reduces to τ_{int}, which is in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ by Definition A.25, or to
$\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\tau'_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$, for $\tau_{\mathsf{int}} \leadsto \tau'_{\mathsf{int}}$,
$\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau'_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$, for $\tau_{\to} \leadsto \tau'_{\to}$,
$\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau'_{\forall};\ \tau_{\forall^+})$, for $\tau_{\forall} \leadsto \tau'_{\forall}$, or
$\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau'_{\forall^+})$, for $\tau_{\forall^+} \leadsto \tau'_{\forall^+}$.
By property 2 of Definition A.17, each of τ'_{int}, $\tau'_{\to}$, $\tau'_{\forall}$, $\tau'_{\forall^+}$ belongs to the same candidate as the respective initial type. Moreover, the length of each of the reducts is k. Therefore, by the inductive hypothesis, each of the reducts is in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$.

Hence $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\vec{\tau})$ always reduces to a type in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. Then by property 3 of Definition A.17, $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \mathsf{int}\ \mathsf{of}\ (\vec{\tau})$ is also in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. Thus $\mathsf{int} \in R_\Omega$. □

LEMMA A.27. $\twoheadrightarrow\ \in R_\Omega \to R_\Omega \to R_\Omega = \mathcal{S}_{\Omega\to\Omega\to\Omega}[\overline{\mathcal{C}}/\overline{\chi}]$.

Proof $\twoheadrightarrow\ \in R_\Omega \to R_\Omega \to R_\Omega$ if for all $\tau_1 \in R_\Omega$ it follows that $(\twoheadrightarrow)\tau_1 \in R_\Omega \to R_\Omega$. This is true if for all $\tau_2 \in R_\Omega$, it follows that $(\twoheadrightarrow)\tau_1\,\tau_2 \in R_\Omega$. Suffices then to prove that $\tau \equiv \mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ (\twoheadrightarrow)\tau_1\,\tau_2\ \mathsf{of}\ (\vec{\tau})$ is in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$, under the conditions in Definition A.25. Since τ_1, τ_2, τ_{int}, $\tau_{\to}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ are strongly normalizable, we can proceed by induction on the length $len(\tau) \equiv \nu(\tau_1)+\nu(\tau_2)+\nu(\tau_{\mathsf{int}}) + \nu(\tau_{\to}) + \nu(\tau_{\forall}) + \nu(\tau_{\forall^+})$.

—$len(\tau) = 0$. The only reduction of τ is to

$$\tau' \equiv \tau_{\rightarrow}\, \tau_1\, (\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \tau_1\ \mathsf{of}\ (\vec{\tau}))$$
$$\tau_2\, (\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \tau_2\ \mathsf{of}\ (\vec{\tau}))$$

Since both τ_1 and τ_2 are in R_Ω, it follows that $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \tau_1\ \mathsf{of}\ (\vec{\tau})$ and $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \tau_2\ \mathsf{of}\ (\vec{\tau})$ are in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. This implies that τ' also belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$.

—$len(\tau) = k + 1$. The case of the head reduction is similar to the previous one. The other possible reductions come from reducing one of the constituent types τ_1, τ_2, τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$; the proofs are similar to the proof of the last case in Lemma A.26.

Since τ is neutral, by property 3 of Definition A.17 $\tau \in \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. □

LEMMA A.28. *If τ is such that for all $\tau_1 \in \mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}]$ we have $\tau\{\tau_1/\alpha\} \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$, then $\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau \in \mathcal{S}_{\kappa_1\rightarrow\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$.*

Proof Consider the neutral type $\tau_0 = (\lambda\alpha : \kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau)\,\tau_1$. We have that τ_1 is strongly normalizable and $\tau\{\alpha'/\alpha\}$ is strongly normalizable. Therefore, τ is also strongly normalizable. We proceed by induction on $len(\tau_0) \equiv \nu(\tau) + \nu(\tau_1)$ to prove that τ_0 always reduces to a type that belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$.

—$len(\tau_0) = 0$. There are two possible reductions. A β_1 reduction yields $\tau\{\tau_1/\alpha\}$, which is by assumption in $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. If $\tau = \tau_0\,\alpha$ and α does not occur free in τ_0, there is an η_1 reduction to $\tau_0\,\tau_1$; but in this case $\tau\{\tau_1/\alpha\} = \tau_0\,\tau_1$.

—$len(\tau_0) = k + 1$. The inductive hypothesis is that for all τ and τ_1, if $\nu(\tau) + \nu(\tau_1) = k$, $\tau_1 \in \mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}]$, and $\tau\{\tau_1/\alpha\} \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$, then $(\lambda\alpha : \kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau)\,\tau_1$ always reduces to a type in $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$.
The β_1 and possible η_1 reductions are handled similarly to the base case. There are two additional reductions. If $\nu(\tau_1) \neq 0$, then τ_0 can reduce to $(\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau)\,\tau_1'$ where $\tau_1 \leadsto \tau_1'$. By property 2 of Definition A.17, τ_1' belongs to $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}]$. Therefore $\tau\{\tau_1'/\alpha\} \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. Moreover, $\nu(\tau) + \nu(\tau_1') = k$. By the inductive hypothesis, $(\lambda\alpha\!:\!\kappa_1.\,\tau)\,\tau_1'$ always reduces to a type that belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. By property 3 of Definition A.17, $(\lambda\alpha\!:\!\kappa_1.\,\tau)\,\tau_1''$ is in $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$.
Alternatively, if $\nu(\tau) \neq 0$, then τ_0 can reduce to $(\lambda\alpha : \kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau')\,\tau_1$ where $\tau \leadsto \tau'$. By Lemma A.13, $\tau\{\tau_1/\alpha\} \leadsto \tau'\{\tau_1/\alpha\}$. By property 2 of Definition A.17, $\tau'\{\tau_1/\alpha\} \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. Moreover, $\nu(\tau') + \nu(\tau_1) = k$. Therefore, by the inductive hypothesis, $(\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau')\,\tau_1$ always reduces to a type in $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. By property 3 of Definition A.17, $(\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau')\,\tau_1$ belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$.

Therefore, the neutral type τ_0 always reduces to a type in $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. By property 3 of Definition A.17, $\tau_0 \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. Therefore, $\lambda\alpha : \kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau$ is in $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}] \rightarrow \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. This implies that $\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau$ belongs to $\mathcal{S}_{\kappa_1\rightarrow\kappa_2}[\overline{\mathcal{C}}/\overline{\chi}]$. □

LEMMA A.29. *$\forall \in \mathcal{S}_{\forall\chi.\,(\chi\rightarrow\Omega)\rightarrow\Omega}[\overline{\mathcal{C}}/\overline{\chi}]$.*

Proof We need to show that for any kind $\kappa_1\{\overline{\kappa}/\overline{\chi}\}$ and corresponding candidate $\mathcal{C}_1$, the type $\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]$ is in $\mathcal{S}_{(\chi\rightarrow\Omega)\rightarrow\Omega}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi]$, or equivalently

$$\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}] \in \mathcal{S}_{\chi\rightarrow\Omega}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi] \rightarrow \mathcal{S}_{\Omega}[\overline{\mathcal{C}}, \mathcal{C}_1/\overline{\chi}, \chi],$$

which follows if for all $\tau \in \mathcal{S}_{\chi\to\Omega}[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$ we have $\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]\,\tau \in \mathcal{S}_\Omega[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$, i.e., $\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]\,\tau \in R_\Omega$. For this to hold, the type

$$\tau' \equiv \mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\;(\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]\,\tau)\;\mathsf{of}\;(\vec{\tau})$$

must be in $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$ whenever the conditions in Definition A.25 are met. Since each of the types τ, τ_{int}, $\tau_{\to}$, $\tau_\forall$, and $\tau_{\forall^+}$ belongs to a candidate, they are strongly normalizable. Thus we can proceed by induction on $len(\tau') \equiv \nu(\tau) + \nu(\tau_{\mathsf{int}}) + \nu(\tau_\to) + \nu(\tau_\forall) + \nu(\tau_{\forall^+})$ to prove τ' always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$.

—$len(\tau') = 0$. Then the only possible reduction of τ' is to $\tau_1' \equiv \tau_\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]\,\tau\,(\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau'')$, where $\tau'' = \mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\;\tau\;\alpha\;\mathsf{of}\;(\vec{\tau})$. For all $\tau_1 \in \mathcal{C}_1$, the type $\tau''\{\tau_1/\alpha\}$ reduces to $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\;\tau\;\tau_1\;\mathsf{of}\;(\vec{\tau})$. By assumption, τ belongs to $\mathcal{S}_\chi[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi] \to \mathcal{S}_\Omega[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$, which is the same set as $\mathcal{C}_1 \to R_\Omega$, hence $\tau\,\tau_1 \in R_\Omega$. This implies $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\;\tau\;\tau_1\;\mathsf{of}\;(\vec{\tau})$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. Therefore, by Lemma A.28 (replacing $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}]$ with $\mathcal{C}_1$ in the lemma), $\lambda\alpha\!:\!\kappa_1\{\overline{\kappa}/\overline{\chi}\}.\,\tau''$ belongs to $\mathcal{C}_1 \to \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$.
  By assumption $\tau_\forall \in \mathcal{S}_{\forall\chi.\,(\chi\to\Omega)\to(\chi\to\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$. Therefore, $\tau_\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]$ is in $\mathcal{S}_{(\chi\to\Omega)\to(\chi\to\kappa)\to\kappa}[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$. This implies that $\tau_\forall\,[\kappa_1\{\overline{\kappa}/\overline{\chi}\}]\,\tau$ is in the set $\mathcal{S}_{(\chi\to\kappa)\to\kappa}[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$. The latter is equal to $\mathcal{S}_{\chi\to\kappa}[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi] \to \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$, which in turn expands to $(\mathcal{C}_1 \to \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]) \to \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}_1/\overline{\chi},\chi]$. But χ does not occur free in κ, so the latter can be written as $(\mathcal{C}_1 \to \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]) \to \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. This implies that τ_1' belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$.

—$len(\tau') = k+1$. The other possible reductions come from the reduction of one of the constituent types τ, τ_{int}, $\tau_\to$, $\tau_\forall$, and $\tau_{\forall^+}$. The proof in this case is similar to the proof of the last case in Lemma A.26.

Since τ' is neutral, by property 3 of Definition A.17 $\tau' \in \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$. □

LEMMA A.30. *If for every kind κ' and reducibility candidate $\mathcal{C}'$ of this kind $\tau\{\kappa'/\chi'\} \in \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$, then $\Lambda\chi'.\,\tau \in \mathcal{S}_{\forall\chi'.\,\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$.*

Proof Consider the neutral type $\tau' = (\Lambda\chi'.\,\tau)\,[\kappa']$ for an arbitrary kind κ'. Since $\tau\{\kappa'/\chi'\}$ is strongly normalizable, τ is strongly normalizable, so we can prove the statement by induction over $\nu(\tau)$, showing that τ' always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$, given that $\tau\{\kappa'/\chi'\} \in \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$.

—$\nu(\tau) = 0$. There are two possible reductions. A β_2 reduction yields $\tau\{\kappa'/\chi'\}$, which is by assumption in $\mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$. If $\tau = \tau_0\,[\chi']$ and χ' does not occur free in τ_0, then the η_2 reduction yields $\tau_0\,[\kappa']$. But in this case $\tau\{\kappa'/\chi'\} = \tau_0\,[\kappa']$.

—$\nu(\tau) = k+1$. There is one additional reduction, $(\Lambda\chi'.\,\tau)\,[\kappa'] \leadsto (\Lambda\chi'.\,\tau_1)\,[\kappa']$, where $\tau \leadsto \tau_1$. By Lemma A.14, we know that $\tau\{\kappa'/\chi'\} \leadsto \tau_1\{\kappa'/\chi'\}$. By property 2 of Definition A.17, $\tau_1\{\kappa'/\chi'\} \in \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$. Moreover, $\nu(\tau_1) = k$. Therefore, by the inductive hypothesis, $(\Lambda\chi'.\,\tau_1)\,[\kappa']$ always reduces to a type in $\mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$. By property 3 of Definition A.17, $(\Lambda\chi'.\,\tau_1)\,[\kappa']$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$.

Therefore, the neutral type τ' always reduces to a type in $\mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$. By property 3 of Definition A.17, $\tau' \in \mathcal{S}_\kappa[\overline{\mathcal{C}},\mathcal{C}'/\overline{\chi},\chi']$. Therefore, $\Lambda\chi'.\,\tau$ belongs to $\mathcal{S}_{\forall\chi'.\,\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$. □

LEMMA A.31. *If* $\tau \in \mathcal{S}_{\forall\chi.\,\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$, *then for every kind* κ' *we have* $\tau\,[\kappa'\{\overline{\kappa}/\overline{\chi}\}] \in \mathcal{S}_{\kappa\{\kappa'/\chi\}}[\overline{\mathcal{C}}/\overline{\chi}]$.

Proof By Definition A.22 $\tau\,[\kappa'\{\overline{\kappa}/\overline{\chi}\}]$ belongs to $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{\chi}, \chi]$, for every kind κ' and reducibility candidate $\mathcal{C}'$ of this kind. Set $\mathcal{C}' = \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$. Applying Lemma A.24 leads to the result. □

LEMMA A.32. $\forall^{+} \in \mathcal{S}_{(\forall\chi.\,\Omega)\to\Omega}[\overline{\mathcal{C}}/\overline{\chi}]$.

Proof We need to show that for all $\tau \in \mathcal{S}_{\forall\chi.\,\Omega}[\overline{\mathcal{C}}/\overline{\chi}]$ we have $\forall^{+}\tau \in R_{\Omega}$. The latter holds if $\tau' \equiv \mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ (\forall^{+}\tau)\ \mathsf{of}\ (\vec{\tau})$ belongs to $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ under the conditions in Definition A.25. We will prove by induction on $len(\tau') \equiv \nu(\tau) + \nu(\tau_{\mathsf{int}}) + \nu(\tau_{\to}) + \nu(\tau_{\forall}) + \nu(\tau_{\forall^{+}})$ that the type $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ (\forall^{+}\tau)\ \mathsf{of}\ (\vec{\tau})$ always reduces to a type in $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$.

—$len(\tau') = 0$. Then the only possible reduction of τ' is to $\tau_{\forall^{+}}\,\tau\,(\Lambda\chi.\,\tau'')$, where $\tau'' = \mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ (\tau\,[\chi])\ \mathsf{of}\ (\vec{\tau})$. For an arbitrary kind κ', $\tau''\{\kappa'/\chi\}$ is equal to $\mathsf{Typerec}[\kappa\{\overline{\kappa}/\overline{\chi}\}]\ \tau\,[\kappa']\ \mathsf{of}\ (\vec{\tau})$. By the assumption on τ, we get that $\tau\,[\kappa'] \in R_{\Omega}$. Therefore, by Definition A.18 $\tau''\{\kappa'/\chi\} \in \mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$. Since χ does not occur free in κ, we can write this as $\tau''\{\kappa'/\chi\} \in \mathcal{S}_{\kappa}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{\chi}, \chi]$ for any candidate $\mathcal{C}'$ of kind κ'. Thus by Lemma A.30 $\Lambda\chi.\,\tau'' \in \mathcal{S}_{\forall\chi.\,\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$. By the assumptions on $\tau_{\forall^{+}}$ and τ, $\tau_{\forall^{+}}\,\tau\,(\Lambda\chi.\,\tau'')$ is in $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$.

—$len(\tau') = k + 1$. The other possible reductions come from the reduction of one of the constituent types τ, τ_{int}, $\tau_{\to}$, $\tau_{\forall}$, and $\tau_{\forall^{+}}$. The proof in this case is similar to the proof of the last case in Lemma A.26.

Since τ' is neutral, by property 3 of Definition A.17, $\tau' \in \mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$. □

We now come to the main result of this section.

THEOREM A.33 (CANDIDACY). *Let* τ *be a type of kind* κ. *Suppose all the free type variables of* τ *are in* $\alpha_1 \ldots \alpha_n$ *of kinds* $\kappa_1 \ldots \kappa_n$ *and all the free kind variables of* κ, $\kappa_1 \ldots \kappa_n$ *are among* $\chi_1 \ldots \chi_m$. *If* $\mathcal{C}_1 \ldots \mathcal{C}_m$ *are candidates of kinds* $\kappa'_1 \ldots \kappa'_m$ *and* $\tau_1 \ldots \tau_n$ *are types of kind* $\kappa_1\{\overline{\kappa'}/\overline{\chi}\} \ldots \kappa_n\{\overline{\kappa'}/\overline{\chi}\}$ *which are in* $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}] \ldots \mathcal{S}_{\kappa_n}[\overline{\mathcal{C}}/\overline{\chi}]$, *then* $\tau\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}$ *belongs to* $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$.

Proof The proof is by induction over the structure of τ.

The cases of int, $\twoheadrightarrow$, $\forall$, $\forall^{+}$ are covered by Lemmas A.26, A.27, A.29, and A.32.

Suppose $\tau = \alpha_i$ and $\kappa = \kappa_i$. Then $\tau\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\} = \tau_i$. By assumption, this belongs to $\mathcal{S}_{\kappa_i}[\overline{\mathcal{C}}/\overline{\chi}]$.

Suppose $\tau = \tau'_1\,\tau'_2$. Then τ'_1 is of kind $\kappa' \to \kappa$ and τ'_2 of kind κ' for some kind κ'. By the inductive hypothesis, $\tau'_1\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}$ belongs to $\mathcal{S}_{\kappa'\to\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$ and $\tau'_2\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}$ belongs to $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$. Therefore, $(\tau'_1\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})\,(\tau'_2\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})$ is in $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$.

Suppose $\tau = \tau'\,[\kappa']$. Then τ' is of kind $\forall\chi_1.\,\kappa_1$ for some χ_1, κ_1; also $\kappa = \kappa_1\{\kappa'/\chi_1\}$. By the inductive hypothesis, $\tau'\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}$ belongs to $\mathcal{S}_{\forall\chi_1.\,\kappa_1}[\overline{\mathcal{C}}/\overline{\chi}]$. By Lemma A.31 $\tau'\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}\,[\kappa'\{\overline{\kappa'}/\overline{\chi}\}]$ is in $\mathcal{S}_{\kappa_1\{\kappa'/\chi_1\}}[\overline{\mathcal{C}}/\overline{\chi}]$, which is equivalent to $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\chi}]$.

Suppose $\tau = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}}; \tau_{\to}; \tau_{\forall}; \tau_{\forall^{+}})$. Then the kinds of τ', τ_{int}, $\tau_{\to}$, $\tau_{\forall}$, and $\tau_{\forall^{+}}$ are Ω, κ, $\Omega \to \kappa \to \Omega \to \kappa \to \kappa$, $\forall\chi.\,(\chi \to \Omega) \to (\chi \to \kappa) \to \kappa$, and

$$
\begin{array}{lrl}
(\textit{type context}) & C ::= & [\,] \mid \Lambda\chi.\,C \mid C\,[\kappa] \mid \lambda\alpha\!:\!\kappa.\,C \mid C\,\tau \mid \tau\,C \\
& & \mid \mathsf{Typerec}[\kappa]\ C \ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \\
& & \mid \mathsf{Typerec}[\kappa]\ \tau \ \mathsf{of}\ (C;\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \mid \mathsf{Typerec}[\kappa]\ \tau \ \mathsf{of}\ (\tau_{\mathsf{int}};\ C;\ \tau_{\forall};\ \tau_{\forall^+}) \\
& & \mid \mathsf{Typerec}[\kappa]\ \tau \ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ C;\ \tau_{\forall^+}) \mid \mathsf{Typerec}[\kappa]\ \tau \ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ C)
\end{array}
$$

Fig. 22. Type contexts

$(\forall\chi.\,\Omega) \rightarrow (\forall\chi.\,\kappa) \rightarrow \kappa$, respectively. By the inductive hypothesis we have

$$
\begin{array}{rcl}
\tau'\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\} & \in & R_\Omega \\
\tau_{\mathsf{int}}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\} & \in & \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}] \\
\tau_{\rightarrow}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\} & \in & \mathcal{S}_{\Omega\rightarrow\kappa\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{\chi}] \\
\tau_{\forall}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\} & \in & \mathcal{S}_{\forall\chi.\,(\chi\rightarrow\Omega)\rightarrow(\chi\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{\chi}] \\
\tau_{\forall^+}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\} & \in & \mathcal{S}_{(\forall\chi.\,\Omega)\rightarrow(\forall\chi.\,\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{\chi}]
\end{array}
$$

Then by definition of R_Ω,

$$
\begin{array}{l}
\mathsf{Typerec}[\kappa\{\overline{\kappa'}/\overline{\chi}\}]\ \tau'\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}\ \mathsf{of}\ (\tau_{\mathsf{int}}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}; \\
\qquad \tau_{\rightarrow}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}; \\
\qquad \tau_{\forall}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\}; \\
\qquad \tau_{\forall^+}\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})
\end{array}
$$

belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\chi}]$.

Suppose $\tau = \lambda\alpha'\!:\!\kappa'.\,\tau_1$. Then τ_1 has some kind κ'' such that $\kappa = \kappa' \rightarrow \kappa''$, and the free type variables of τ_1 are in $\alpha_1, \ldots, \alpha_n, \alpha'$. By the inductive hypothesis, $\tau_1\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}, \tau'/\overline{\alpha}, \alpha'\}$ is in $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}/\overline{\chi}]$, where τ' is of kind $\kappa'\{\overline{\kappa'}/\overline{\chi}\}$ and belongs to $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{\chi}]$. This implies that $(\tau_1\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})\{\tau'/\alpha'\}$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}/\overline{\chi}]$ (since α' occurs free only in τ_1). By Lemma A.28, $\lambda\alpha' : \kappa'\{\overline{\kappa'}/\overline{\chi}\}.\,(\tau_1\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})$ is in $\mathcal{S}_{\kappa'\rightarrow\kappa''}[\overline{\mathcal{C}}/\overline{\chi}]$.

Suppose $\tau = \Lambda\chi'.\,\tau'$. Then the kind of τ' is κ'', and $\kappa = \forall\chi'.\,\kappa''$. By the inductive hypothesis, $\tau'\{\overline{\kappa'}, \kappa'/\overline{\chi}, \chi'\}\{\overline{\tau}/\overline{\alpha}\}$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{\chi}, \chi']$ for an arbitrary kind κ' and candidate $\mathcal{C}'$ of kind κ'. Since χ' occurs free only in τ', we get that $(\tau'\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})\{\kappa'/\chi'\}$ is in $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{\chi}, \chi']$. By Lemma A.30 the type $\Lambda\chi'.\,(\tau'\{\overline{\kappa'}/\overline{\chi}\}\{\overline{\tau}/\overline{\alpha}\})$ is in $\mathcal{S}_{\forall\chi'.\,\kappa''}[\overline{\mathcal{C}}/\overline{\chi}]$. □

Suppose SN_i is the set of strongly normalizable types of kind κ_i.

COROLLARY A.34. *All types are strongly normalizable.*

Proof Follows from Theorem A.33 by setting $\mathcal{C}_i = SN_i$ and $\tau_i = \alpha_i$. □

A.3 Confluence in the λ_i^ω Type Language

To prove confluence of the reduction in the type language of λ_i^ω, we first define the compatible extension $\mapsto$ of the one-step reduction $\rightsquigarrow$. Let the set of type contexts (ranged over by C) be defined inductively as shown in Figure 22. A context is thus a "type term" with a hole $[\,]$; the term $C\{\tau\}$ is defined as the type obtained by replacing the hole in C by τ.

DEFINITION A.35. $\tau_1 \mapsto \tau_2$ iff there exist types τ_1' and τ_2' and a type context C such that $\tau_1 = C\{\tau_1'\}$, $\tau_2 = C\{\tau_2'\}$, and $\tau_1' \rightsquigarrow \tau_2'$.

Let as usual $\mapsto^*$ denote the reflexive and transitive closure of $\mapsto$.

LEMMA A.36. *If $\tau \mapsto \tau'$, then $C\{\tau\} \mapsto C\{\tau'\}$.*

Proof From compositionality of contexts, i.e., since for all contexts C_1 and C_2 and types τ, $C_1\{C_2\{\tau\}\} = C\{\tau\}$ for some context C, which is constructed inductively on the structure of C_1. □

COROLLARY A.37. *If $\tau \mapsto^* \tau'$, then $C\{\tau\} \mapsto^* C\{\tau'\}$.*

The following lemmas are proved by induction on the structure of contexts.

LEMMA A.38. *If $\tau_1 \mapsto \tau_2$, then $\tau_1\{\tau/\alpha\} \mapsto \tau_2\{\tau/\alpha\}$.*

Proof Sketch Follows from Lemma A.13. □

LEMMA A.39. *If $\tau_1 \mapsto \tau_2$, then $\tau_1\{\kappa/\chi\} \mapsto \tau_2\{\kappa/\chi\}$.*

Proof Sketch Follows from Lemma A.14. □

LEMMA A.40. *If $\mathcal{E}; \Delta \vdash C\{\tau\} : \kappa$, then there exist $\mathcal{E}'$, Δ', and κ' such that $\mathcal{E}'; \Delta' \vdash \tau : \kappa'$; furthermore, if $\mathcal{E}'; \Delta' \vdash \tau' : \kappa'$, then $\mathcal{E}; \Delta \vdash C\{\tau'\} : \kappa$.*

By induction on the structure of types we prove the following substitution lemmas.

LEMMA A.41. *If $\mathcal{E}; \Delta, \alpha : \kappa' \vdash \tau : \kappa$ and $\mathcal{E}; \Delta \vdash \tau' : \kappa'$, then $\mathcal{E}; \Delta \vdash \tau\{\tau'/\alpha\} : \kappa$.*

LEMMA A.42. *If $\mathcal{E}, \chi; \Delta \vdash \tau : \kappa$ and $\mathcal{E} \vdash \kappa'$, then $\mathcal{E}; \Delta\{\kappa'/\chi\} \vdash \tau\{\kappa'/\chi\} : \kappa\{\kappa'/\chi\}$.*

Now we can show subject reduction for $\leadsto$.

LEMMA A.43. *If $\mathcal{E}; \Delta \vdash \tau : \kappa$ and $\tau \leadsto \tau'$, then $\mathcal{E}; \Delta \vdash \tau' : \kappa$.*

Proof Sketch Follows by case analysis of the reduction relation $\leadsto$ and the substitution Lemmas A.41 and A.42. □
Then we have subject reduction for $\mapsto$ as a corollary of Lemmas A.43 and A.40.

COROLLARY A.44. *If $\mathcal{E}; \Delta \vdash \tau : \kappa$ and $\tau \mapsto \tau'$, then $\mathcal{E}; \Delta \vdash \tau' : \kappa$.*

For our confluence proof we need another property of substitution.

LEMMA A.45. *If $\tau_1 \mapsto \tau_2$, then $\tau\{\tau_1/\alpha\} \mapsto^* \tau\{\tau_2/\alpha\}$.*

Proof The proof is by induction on the structure of τ. The cases when τ is a constant, $\tau = \alpha$, or $\tau = \beta \neq \alpha$, are straightforward.
case $\tau = \Lambda\chi.\,\tau'$: W.l.o.g. assume that χ is not free in τ_1, so that $\tau\{\tau_1/\alpha\} = \Lambda\chi.\,(\tau'\{\tau_1/\alpha\})$; then by subject reduction (Corollary A.44) χ is not free in τ_2, hence $\tau\{\tau_2/\alpha\} = \Lambda\chi.\,(\tau'\{\tau_2/\alpha\})$. By the induction hypothesis we have that $\tau'\{\tau_1/\alpha\} \mapsto^* \tau'\{\tau_2/\alpha\}$. Then by Corollary A.37 for the context $\Lambda\chi.\,[\,]$ we obtain $\Lambda\chi.\,(\tau'\{\tau_1/\alpha\}) \mapsto^* \Lambda\chi.\,(\tau'\{\tau_2/\alpha\})$.
The cases of $\tau = \tau'\,[\kappa]$ and $\tau = \lambda\beta : \kappa.\,\tau'$ are similar.
case $\tau = \tau'\,\tau''$: By induction hypothesis we have

$$\begin{array}{ll}(1) & \tau'\{\tau_1/\alpha\} \mapsto^* \tau'\{\tau_2/\alpha\} \\ (2) & \tau''\{\tau_1/\alpha\} \mapsto^* \tau''\{\tau_2/\alpha\}.\end{array}$$

Using context $[\,]\,(\tau''\{\tau_1/\alpha\})$, from (1) and Corollary A.37 it follows that

$$\tau\{\tau_1/\alpha\} = (\tau'\{\tau_1/\alpha\})\,(\tau''\{\tau_1/\alpha\}) \mapsto^* (\tau'\{\tau_2/\alpha\})\,(\tau''\{\tau_1/\alpha\});$$

then using context $(\tau'\{\tau_2/\alpha\})\,[\,]$, from (2) and Corollary A.37 we have

$$(\tau'\{\tau_2/\alpha\})\,(\tau''\{\tau_1/\alpha\}) \mapsto^* (\tau'\{\tau_2/\alpha\})\,((\tau''\{\tau_2/\alpha\})) = \tau\{\tau_2/\alpha\}$$

and the result follows since $\mapsto^*$ is closed under transitivity.
The case of $\tau = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$ is similar. □

The next step is to prove local confluence of the reduction of well-formed types.

LEMMA A.46. *If* $\mathcal{E};\Delta \vdash \tau : \kappa_0$, $\tau \mapsto \tau_1$, *and* $\tau \mapsto \tau_2$, *then there exists* τ_0 *such that* $\tau_1 \mapsto^* \tau_0$ *and* $\tau_2 \mapsto^* \tau_0$.

Proof The proof proceeds by induction on the structure of the derivation of $\mathcal{E};\Delta \vdash \tau : \kappa_0$. For the base cases, corresponding to τ being one of the Ω constructors or a type variable, no rules of reduction apply, so the result is trivial. For the other cases, let $C_1, C_2, \tau_1', \tau_2', \tau_1''$, and τ_2'' be such that $\tau = C_1\{\tau_1'\} = C_2\{\tau_2'\}$, $\tau_1 = C_1\{\tau_1''\}$, $\tau_2 = C_2\{\tau_2''\}$, and $\tau_1' \leadsto \tau_1''$, $\tau_2' \leadsto \tau_2''$.
case $\tau = \Lambda\chi.\,\tau'$: An inspection of the definition of contexts shows that the only possible forms for C_1 and C_2 are $[\,]$ and $\Lambda\chi.\,C$. Thus, accounting for the symmetry, there are the following three subcases:

—Both C_1 and C_2 are $[\,]$. The only reduction rule that applies then is η_2, so $\tau_1 = \tau_2$.

—$C_1 = \Lambda\chi.\,C_1'$ and $C_2 = \Lambda\chi.\,C_2'$. Then the result follows by the inductive hypothesis and Corollary A.37.

—$C_1 = [\,]$ and $C_2 = \Lambda\chi.\,C_2'$. Again. the only reduction for τ_1' is η_2, so $\tau' = \tau''\,[\chi]$ for some τ''. Then there are two cases for τ_2'. First, if $C_2' = [\,]$, then $\tau_2' = \tau'$, and—by inspection of the rules—in the case of kind application the only possible reduction is via β_2, hence $\tau'' = \Lambda\chi'.\,\tau'''$ for some χ' and τ'''. Representing the reductions diagrammatically, we have immediate confluence (up to renaming of bound variables):

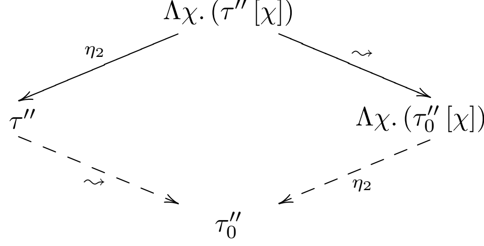
$$\Lambda\chi.\,((\Lambda\chi'.\,\tau''')\,[\chi])$$

$$\eta_2 \swarrow \qquad \searrow \beta_2$$

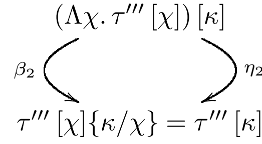
$$\Lambda\chi'.\,\tau''' =_\alpha \Lambda\chi.\,\tau'''\{\chi/\chi'\}$$

The second case accounts for all other possibilities for C_2' (which must be of the form $C_2''\,[\chi]$) and reduction rules that can be applied in $\tau'' = C_2''\{\tau_2'\}$ to reduce it (by assumption) to $C_2''\{\tau_2''\}$, which we denote by τ_0''. The dashed

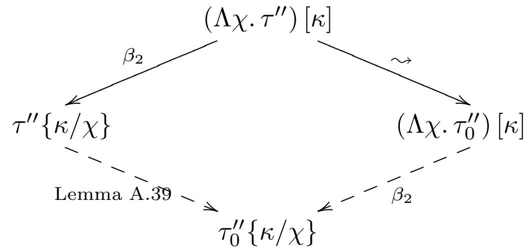
arrows show the reductions that complete local confluence.



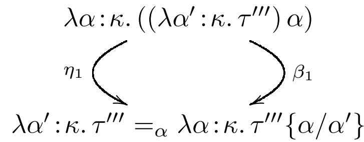
case $\tau = \tau'\,[\kappa]$: Again by inspection of the rules we have that the contexts are either empty or of the form $C\,[\kappa]$. The symmetric cases are handled as in the case of kind abstraction above. The interesting situation is when $C_1 = [\,]$ and $C_2 = C'_2\,[\kappa]$. The only reduction rule that applies for τ_1 is then β_2, hence $\tau' = \Lambda\chi.\,\tau''$ for some χ and τ''. Again we have two major cases for τ_2: first, if $C'_2 = [\,]$, only η_2 applies, so $\tau'' = \tau'''\,[\chi]$ for some τ''', thus



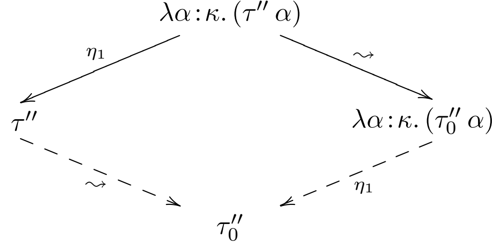
In all other cases we have $C'_2 = \Lambda\chi.\,C''_2$, so $\tau'' = C''_2\{\tau'_2\} \mapsto C''_2\{\tau''_2\}$; letting τ''_0 stand for the latter, we have the diagram



case $\tau = \lambda\alpha:\kappa.\,\tau'$: The contexts can be either empty or of the form $\lambda\alpha:\kappa.\,C$. The symmetric cases are similar to those above. In the case when $C_1 = [\,]$ and $C_2 = \lambda\alpha:\kappa.\,C'_2$, the only rule that applies for the reduction of τ'_1 is η_1, so $\tau' = \tau''\,\alpha$ for some τ''. Again, there are two cases for τ'_2: First, if $C'_2 = [\,]$, we have $\tau'_2 = \tau' = \tau''\,\alpha$, and the only reduction rule for application is β_1, hence $\tau'' = \lambda\alpha' : \kappa'.\,\tau'''$ for some α', κ', and τ'''. Since $\mathcal{E};\Delta \vdash \tau : \kappa_0$, the subterm $(\lambda\alpha' : \kappa'.\,\tau''')\,\alpha$ must be well-typed in an environment assigning kind κ to α, hence $\kappa' = \kappa$, so that



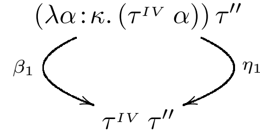
In all other cases for C_2' (which are of the form $C_2''\,\alpha$), we have $\tau'' = C_2''\{\tau_2'\} \mapsto C_2''\{\tau_2''\}$; denoting the latter type by τ_0'', we obtain



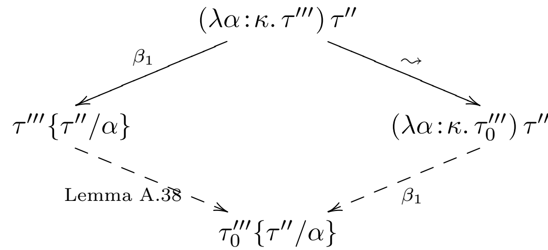
case $\tau = \tau'\,\tau''$: There are three possibilities for the contexts C_1 and C_2: to be empty, of the form $C\,\tau'$, or of the form $\tau\,C$. The symmetric cases proceed as before.

When $C_1 = C_1'\,\tau''$ and $C_2 = \tau'\,C_2'$, the redexes in τ_1' and τ_2' are in different subterms of the type, hence the reductions commute: we have $C_1'\{\tau_1'\} = \tau'$ and $C_2'\{\tau_2'\} = \tau''$, therefore $\tau_1 = (C_1'\{\tau_1''\})\,(C_2'\{\tau_2'\})$ and $\tau_2 = (C_1'\{\tau_1'\})\,(C_2'\{\tau_2''\})$, which both reduce to $(C_1'\{\tau_1''\})\,(C_2'\{\tau_2''\})$.

When $C_1 = [\,]$ and $C_2 = C_2'\,\tau''$, the only reduction rule that applies for $\tau_1' = \tau'\,\tau''$ is β_1, hence $\tau' = \lambda\alpha\!:\!\kappa.\,\tau'''$ for some α, κ, and τ'''. As before, there are two cases for C_2'. If it is empty, then the only reduction rule that applies to $\tau_2' = \tau'$ is η_1, hence $\tau''' = \tau^{IV}\,\alpha$ for some τ^{IV}, and local confluence follows by

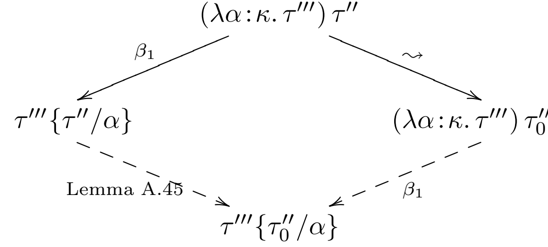


Alternatively, C_2' must be of the form $\lambda\alpha : \kappa.\,C_2''$, where $C_2''\{\tau_2'\} = \tau'''$. Then $\tau''' \mapsto C_2''\{\tau_2''\} \equiv \tau_0'''$, and we have



When $C_1 = [\,]$ and $C_2 = \tau'\,C_2'$, again the only reduction rule that applies for $\tau_1' = \tau'\,\tau''$ is β_1, so $\tau' = \lambda\alpha\!:\!\kappa.\,\tau'''$ for some α, κ, and τ'''. This time, regardless of

the structure of C_2', we have that $\tau'' = C_2''\{\tau_2'\} \mapsto C_2''\{\tau_2''\} \equiv \tau_0''$, hence

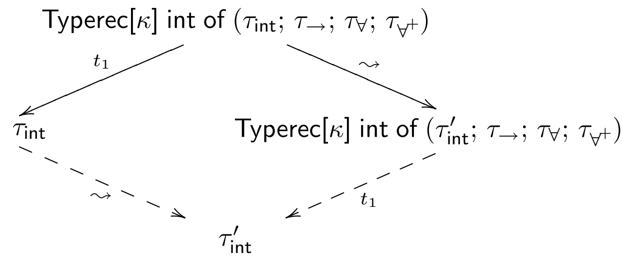


case $\tau = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$: The contexts can be empty or of the forms

$$\begin{array}{l}
\mathsf{Typerec}[\kappa]\ C\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \\
\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (C;\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \\
\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ C;\ \tau_{\forall};\ \tau_{\forall^+}) \\
\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ C;\ \tau_{\forall^+}) \\
\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ C)
\end{array}$$

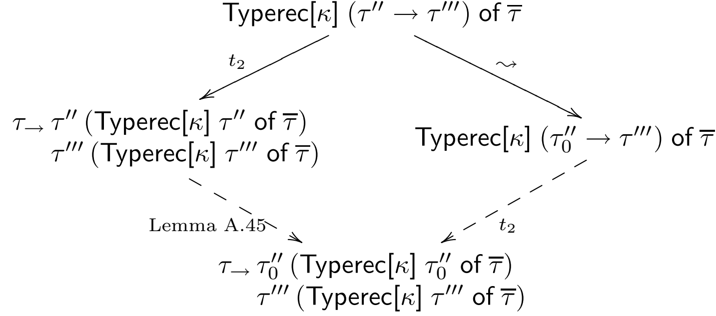
The symmetric cases and the non-overlapping cases are handled as before. Accounting for the symmetry, the remaining cases are when $C_1 = [\,]$ and C_2 is not empty. Then the reduction rule for τ_1' must be one of $t_1, t_2, t_3,$ and t_4. Since there is no η rule for Typerec, the proofs are straightforward.

subcase t_1: then $\tau' = \mathsf{int}$. The result of the reduction under C_2 is ignored and local confluence is trivial, unless $C_2 = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (C_2';\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$. In the latter case,



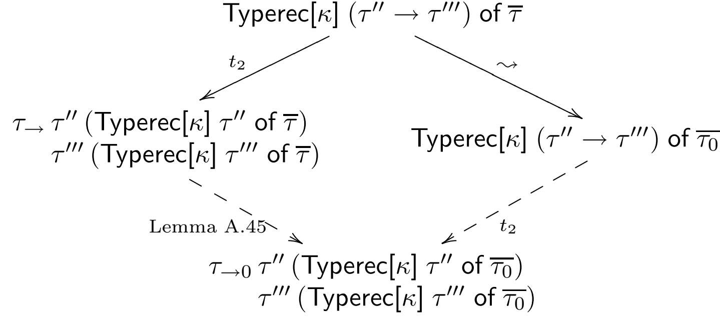
subcase t_2: then $\tau' = \tau'' \rightarrow \tau'''$. We will use $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ \overline{\tau}$ as a shorthand for $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$, and similarly for contexts. If $C_2 = \mathsf{Typerec}[\kappa]\ C_2'\ \mathsf{of}\ \overline{\tau}$, then there are two subcases for C_2' (which must have the $\rightarrow$

constructor at its head). Thus, if $C'_2 = C''_2 \to \tau'''$,



where $\tau'' = C''_2\{\tau'_2\} \mapsto C''_2\{\tau''_2\} \equiv \tau''_0$. The case of $C'_2 = \tau'' \to C''_2$ is similar.

Of the other cases we will only show the reduction in the position of $\tau_\to$, writing $\overline{\tau_0}$ for $(\tau_{\mathsf{int}};\, \tau_{\to 0};\, \tau_\forall;\, \tau_{\forall^+})$, where $\tau_\to \mapsto \tau_{\to 0}$.



subcases t_3 and t_4 are similar to t_2. □

COROLLARY A.47. *If* $\mathcal{E};\Delta \vdash \tau : \kappa$, $\tau \mapsto^* \nu$, *and* $\tau \mapsto^* \tau'$, *then* $\tau' \mapsto^* \nu$.

THEOREM A.48. *If* $\mathcal{E};\Delta \vdash \tau : \kappa$, *then there exists exactly one* ν *such that* $\tau \mapsto^* \nu$.

Proof From Corollaries A.34 and A.47. □

ACKNOWLEDGMENT

We thank the anonymous referees for their careful reading and many insightful comments.

REFERENCES

CRARY, K. AND WEIRICH, S. 1999. Flexible type analysis. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 233–248.

CRARY, K., WEIRICH, S., AND MORRISETT, G. 1998. Intensional polymorphism in type-erasure semantics. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 301–312.

DESPEYROUX, J., PFENNING, F., AND SCHÜRMANN, C. 1997. Primitive recursion for higher-order abstract syntax. In *Proceedings of the 3rd International Conference on Typed Lambda Calculus and Applications (TLCA'97)*. Lecture Notes in Computer Science, vol. 1210. Springer-Verlag, Nancy, France, 147–163.

DUBOIS, C., ROUAIX, F., AND WEIS, P. 1995. Extensional polymorphism. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 118–129.

DUGGAN, D. 1998. A type-based semantics for user-defined marshalling in polymorphic languages. In *Proceedings of the 1998 International Workshop on Types in Compilation*. Lecture Notes in Computer Science, vol. 1473. Springer-Verlag, Kyoto, Japan, 273–298.

GIRARD, J. Y. 1972. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Ph.D. thesis, University of Paris VII.

GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. 1989. *Proofs and Types*. Cambridge University Press.

HARPER, R. AND LILLIBRIDGE, M. 1993. Explicit polymorphism and CPS conversion. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 206–219.

HARPER, R. AND MITCHELL, J. C. 1999. Parametricity and variants of Girard's J operator. *Information Processing Letters 70,* 1 (Apr.), 1–5.

HARPER, R. AND MORRISETT, G. 1995. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 130–141.

LEAGUE, C., SHAO, Z., AND TRIFONOV, V. 1999. Representing Java classes in a typed intermediate language. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 183–196.

MINAMIDE, Y. 1997. Full lifting of type parameters. Technical report, RIMS, Kyoto University.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 271–283.

MONNIER, S., SAHA, B., AND SHAO, Z. 2001. Principled scavenging. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 81–91.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998. From System F to typed assembly language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 85–97.

NECULA, G. C. 1998. Compiling with proofs. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Tech Report CMU-CS-98-154.

OHORI, A. AND KATO, K. 1993. Semantics for communication primitives in a polymorphic language. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 99–112.

PETERSON, J. AND JONES, M. 1993. Implementing type classes. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 227–236.

PFENNING, F. AND PAULIN-MOHRING, C. 1989. Inductively defined types in the calculus of constructions. In *Proceedings of the 5th Conference on the Mathematical Foundations of Programming Semantics*. Springer-Verlag, New Orleans, Louisiana, 209–228.

PIERCE, B., DIETZEN, S., AND MICHAYLOV, S. 1989. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University.

REYNOLDS, J. C. 1974. Towards a theory of type structure. In *Proceedings of the Colloque sur la Programmation*. Lecture Notes in Computer Science, vol. 19. Springer-Verlag, Berlin, 408–425.

SHAO, Z. 1997a. Flexible representation analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 85–98.

SHAO, Z. 1997b. An overview of the FLINT/ML compiler. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation*.

SHAO, Z. 1998. Typed cross-module compilation. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

SHAO, Z. 1999. Transparent modules with fully syntactic signatures. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 220–232.

SHAO, Z. AND APPEL, A. W. 1995. A type-based compiler for Standard ML. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 116–129.

SHAO, Z., SAHA, B., TRIFONOV, V., AND PAPASPYROU, N. 2002. A type system for certified binaries. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 217–232.

TARDITI, D. 1996. Design and implementation of code optimizations for a type-directed compiler for Standard ML. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Tech Report CMU-CS-97-108.

TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. 1996. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 181–192.

WEIRICH, S. 2000. Type-safe cast. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 58–67.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1 (Nov.), 38–94.

YANG, Z. 1998. Encoding types in ML-like languages. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 289–300.

Received November 2000; revised December 2001 and May 2002; accepted July 2002.