Intensional Analysis of Higher-Kinded Recursive Types[⋆]

Technical Report YALEU/DCS/TR-1240

Gregory D. Collins   Zhong Shao

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

Abstract. Recursive types are ubiquitous in modern programming languages, as they occur when typing constructs such as datatype definitions. Any type-theoretic framework must effectively deal with recursive types if it purports to be applicable to real languages such as ML and Haskell. Intensional Type Analysis [1] and Certified Binaries [2] are two such type-theoretic frameworks. Previous work in these areas, however, has not adequately supported recursive types. In this paper we present a new formulation of recursive types which subsumes the traditional one. We show that intensional analysis over these types (including higher-kinded recursive types) can be supported via inductive elimination. Our solution is simple, general, and extensible; the typing rules for higher-kinded recursive types are concise and very easy to understand.

1 Introduction

Recursive types are ubiquitous in modern programming languages, as they occur when typing constructs such as ML-like datatype definitions. Recursive data structures can be mutually recursive, generic (i.e., polymorphic), or type-dependent [3]; typing them would require the use of higher-kinded recursive types. Any type-theoretic framework must effectively deal with all forms of recursive types if it purports to be applicable to a language such as Standard ML (SML) [4] or Haskell [5].

Intensional Type Analysis (ITA) [1] and Certified Binaries (CB) [2] are two such type-theoretic frameworks. Intensional type analysis is a framework for reasoning (either at the type level or the term level) about the structure of types in a program. Certified binaries is a framework for reasoning about advanced properties on programs written in typed intermediate or assembly languages.

Previous work [2, 6, 7] has attempted to solve these problems by defining the type constructors of term languages as constructors of an *inductive* kind Ω. Defining types this way allows the use of inductive elimination to reason about the structure of types.

⋆ This work is supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-0081590, and NSERC fellowship PGSA-242848-2001. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

Using inductive elimination is a natural way to think about doing ITA, as well as enabling one to reason about complex propositions and proofs of properties of programs, which is necessary for proof-carrying code [8]. Previous work has been able to represent *most* types used by a modern programming language such as Standard ML (SML) using these inductive definitions [7].

Both ITA and CB are designed to reason about the properties of programs written in a modern programming language such as SML. Thusfar, however, all of the work in this area of which we are aware (for example [1, 2, 6, 7, 9]) has either completely ignored or poorly supported recursive types.

In this paper we will show a formulation of the recursive type constructor μ which can be defined inductively. We show that this approach is more general than the standard formulation of recursive types and how SML-style datatypes may be represented in our system. We also show how our scheme can be used to support intensional analysis of higher-kinded recursive types.

Because our scheme allows the inductive definition of the μ type constructor, we are able to use inductive elimination to reason about the structure of recursive types. In contrast to other type-theoretic formulations of recursive types, it works over type terms of higher kinds. This means that in the context of a certified binary (see, for instance, Shao et al. [2]), we could reason about recursive types over kinds like $\mathsf{Nat} \to \Omega$. Our solution to this problem also has the advantage of elegance and simplicity; the typing rules for our μ constructor are small and very easy to understand.

1.1 Recursive Types

The standard approach to recursive types in the context of modern programming languages such as SML (as presented in Harper and Stone [10]) is to present a higher-kinded type constructor $\mu : (\kappa \to \kappa) \to \kappa$. The unrolling of a recursive type $\mu(\lambda t : \kappa.\tau)$ is then $\tau[t/\mu(\lambda t : \kappa.\tau)]$. Unfortunately (unless $\kappa = \Omega$), the result of applying this type constructor μ does not belong to the "ground kind" Ω, which we define as the kind of all type constructors of terms. In other words, in general we cannot type an expression in the term language with a recursive type. Instead, we would employ Harper/Stone's solution, which is to use a "projection" operator to take the result back to Ω. Consider the following simple example (in SML syntax):

$$\begin{array}{rl} \texttt{datatype} \quad \texttt{tree} = & \texttt{Leaf of unit} \\ & |\, \texttt{Node} \;\; \texttt{of forest} * \texttt{int} \\ \texttt{and forest} = & \texttt{Trees of tree} * \texttt{tree} \end{array} \tag{1.1}$$

In a standard presentation, since the types mutually recurse, we would package them together using a *fixpoint* function:

$$\lambda(\texttt{tree}, \texttt{forest}) : \Omega \circledast \Omega.\ \langle \mathsf{unit} + (\texttt{forest}(t) * \mathsf{int}), \texttt{tree}(t) * \texttt{tree}(t) \rangle\ .$$

Applying the recursive type constructor μ (and cleaning up the syntax), the type of this definition then becomes:

$$\tau := \mu(\lambda t : (\Omega \circledast \Omega).\ \langle \mathsf{unit} + (\pi_2(t) * \mathsf{int}), \pi_1(t) * \pi_1(t) \rangle).$$

We could then project this result back to obtain the type of `tree` as $\pi_1(\tau)$ and the type of `forest` as $\pi_2(\tau)$.

This approach is not especially general; in particular, in [10], projections of $\kappa_1 \Rightarrow \kappa_2$ back to Ω are handled differently in the static semantics from projections of $\kappa_1 \circledast \kappa_2$, and it is not clear how their mechanism may be applied to higher kinds. This is not surprising, since the system was designed to be just powerful enough to formalize SML datatypes.

Our solution is to explicitly combine the fixpoint function and the projection operation into one type, writing a recursive type over κ as $\mu_\kappa(f, g)$. For instance, if we let $F = \lambda t : (\Omega \circledast \Omega).\ \langle \mathsf{unit} + (\pi_2(t) * \mathsf{int}), \pi_1(t) * \pi_1(t)\rangle$ as above, then we can write `tree` as

$$\mu_{\Omega \circledast \Omega}(F, \lambda t : (\Omega \circledast \Omega).\pi_1(t)),$$

and `forest` as

$$\mu_{\Omega \circledast \Omega}(F, \lambda t : (\Omega \circledast \Omega).\pi_2(t)).$$

Our μ can thus be viewed as a constructor

$$\mu : \Pi k : \mathsf{Set}.(k \to k) \to (k \to \Omega) \to \Omega.$$

We note that this definition is inductive in Ω. Using this definition has several benefits. It is general, it fits into a comprehensive framework of typechecking using inductive definitions, and it allows inductive reasoning on recursive types. Key operations such as roll/unroll can be expressed using type-level reasoning, by way of inductive elimination. We will show that this formulation of recursive types is at least as general as the standard one.

2 A Language using Traditional Recursive Types: λ_{M1}

We will begin by showing a minimal language which uses a more traditional formulation of recursive types. We'll call this language λ_{M1}. To simplify our analysis we only consider constants representing natural numbers (n). We present only pairs, not tuples (tuples may be defined as a "syntactic sugar" on pairs). The term language also lacks function polymorphism and existential types. See Figure 1 for the syntax of this language.

A ground-level type (i.e., a type which we can type terms with) τ in λ_{M1} can be the integer type (int), the unit type (unit), the function type ($\tau_1 \to \tau_2$), the sum type ($\tau_1 + \tau_2$), or the tuple type ($\tau_1 * \tau_2$). In addition, λ_{M1} has higher-order types such as lambda abstraction ($\lambda t : \kappa.\tau$), type variables t, and type tuples ($\langle \tau_1, \tau_2 \rangle$). The type system of λ_{M1} essentially contains a computation language consisting of a simply-typed lambda calculus; thus we also have type-function application ($\tau_1\ \tau_2$) and type-tuple projection $\pi_i(\tau)$. We (of course) also have the recursive type $\mu t : \kappa.\tau$, which will be explained later. The types of λ_{M1} are kinded; Ω is the kind of all ground terms, $\kappa_1 \Rightarrow \kappa_2$ is the kind of type-functions, and $\kappa_1 \circledast \kappa_2$ is the kind of type-tuples.

The term language of λ_{M1} is completely standard: we have variables (x), integer constants (n), if statements, function definitions and applications, tuple creation and projection, sum-type injection (inl, inr) and elimination (case), and recursive roll and unroll.

$$
\begin{array}{lll}
\text{(kinds)} & \kappa ::= \Omega \mid \kappa_1 \Rightarrow \kappa_2 \mid \kappa_1 \circledast \kappa_2 \\[1ex]
\text{(types)} & \tau ::= \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \mid \mu t : \kappa.\tau \\
 & \quad \mid t \mid \lambda t : \kappa.\tau \mid \tau_1(\tau_2) \mid \pi_i(\tau) \mid \langle \tau_1, \tau_2 \rangle \\[1ex]
\text{(expr)} & e ::= x \mid n \mid op(e_1, e_2) \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \\
 & \quad \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid e_1(e_2) \mid (e_1, e_2) \mid () \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
 & \quad \mid \mathsf{roll}_\tau(e) \mid \mathsf{unroll}(e) \mid \mathsf{inl}_{\tau_1+\tau_2}(e_1) \mid \mathsf{inr}_{\tau_1+\tau_2}(e_2) \\
 & \quad \mid \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inr}(x : \tau_2) \Rightarrow e_2\ \mathsf{end} \\[1ex]
\text{(oper)} & op ::= + \mid - \mid \ldots
\end{array}
$$

Fig. 1. A lightweight language using traditional μ: λ_{M1}

type environment	Δ	well-formed types	$\Delta \vdash \tau : \kappa$
term environment	Γ	type-equivalence	$\Delta \vdash \tau_1 \equiv \tau_2 : \kappa$
well-formed Δ	$\vdash \Delta$ *type env*	well-formed term	$\Gamma; \Delta \vdash e : \tau$
well-formed Γ	$\Delta \vdash \Gamma$ *term env*		
well-formed kind	$\vdash \kappa$ *kind*	term evaluation	$e_1 \hookrightarrow e_2$

Fig. 2. Summary of formal properties of λ_{M1}

For λ_{M1} we use a standard small-step operational semantics. The values of the language are as follows:

$$
\begin{array}{ll}
\text{(values)}\ v ::= & n \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid \mathsf{inl}_{\tau_1+\tau_2}(v) \mid \mathsf{inr}_{\tau_1+\tau_2}(v) \\
 & \mid () \mid (v_1, v_2) \mid \mathsf{roll}_\tau(v)
\end{array}
$$

Figure 2 contains a summary of the formal judgments of λ_{M1}. For the complete formal properties of λ_{M1}, including dynamic semantics, static semantics, and type-safety theorems, see Appendix A.

2.1 Recursive Types

This language uses a standard presentation of recursive types; that is, μ has the form $\mu t : \kappa.\tau$, where τ ends up having kind κ. The typing judgments for roll and unroll are as follows: (taken from Harper and Stone [10])

$$\frac{\Delta, t : \kappa \vdash \tau : \kappa}{\Delta \vdash \mu t : \kappa.\tau : \kappa} \qquad \text{(Well-formed } \mu\text{)}$$

$$\frac{\begin{array}{c}\mathsf{p} = \pi_i \mid \cdot \\ \mathsf{s} = \tau'' \mid \cdot \\ \Delta \vdash \tau \equiv (\mathsf{p}(\mu t : \kappa.\tau'))(\mathsf{s}) : \Omega \\ \Gamma; \Delta \vdash e : \tau\end{array}}{\Gamma; \Delta \vdash \mathsf{unroll}(e) : (\mathsf{p}(\tau'[t := \mu t : \kappa.\tau']))(\mathsf{s})} \quad \text{(Unroll)}$$

$$\frac{\begin{array}{c}\mathsf{p} = \pi_i \mid \cdot \\ \mathsf{s} = \tau'' \mid \cdot \\ \Delta \vdash \tau \equiv (\mathsf{p}(\mu t : \kappa.\tau'))(\mathsf{s}) : \Omega \\ \Gamma; \Delta \vdash e : (\mathsf{p}(\tau'[t := \mu t : \kappa.\tau']))(\mathsf{s})\end{array}}{\Gamma; \Delta \vdash \mathsf{roll}_\tau(e)} \quad \text{(Roll)}$$

In the rules above, p and s are "optional", and are required to coerce the μ-type back to Ω. The fact that they are necessary indicates a fault in this formulation of recursive types; we have to resort to "tricks" such as this to typecheck roll and unroll. A worse problem is that these rules only work for a limited set of kinds; it is completely unclear how to extend this mechanism to be more general.

The roll and unroll terms act as *coercions* on the typing of variables; the values end up unchanged. This is evidenced by the following rule taken from the operational semantics of λ_{M1}:

$$\frac{}{\mathsf{unroll}(\mathsf{roll}(v)) \hookrightarrow v}$$

2.2 Examples

We will now consider some illustrative examples. We'll relax our formalism for a moment and extend λ_{M1} to include tuples of arbitrary size. Consider the following set of datatypes (from Okasaki [13]):

$$\begin{array}{rl} \texttt{datatype } \alpha\ \texttt{Quad} = & \texttt{Q of } \alpha * \alpha * \alpha * \alpha \\ \texttt{and } \alpha\ \texttt{Square} = & \texttt{Zero of } \alpha \\ & \mid \texttt{Succ of } (\alpha\ \texttt{Quad})\ \texttt{Square} \end{array} \tag{2.1}$$

We can type this datatype (erasing the constructor names) as:

$$\begin{aligned} \tau = & \mu t : (\Omega \Rightarrow \Omega) \circledast (\Omega \Rightarrow \Omega). \\ & \langle \lambda\alpha : \Omega.(\alpha * \alpha * \alpha * \alpha), \lambda\alpha : \Omega.(\alpha + (\pi_2(t)\ (\pi_1(t)\ \alpha)) \rangle \end{aligned}$$

In order to create a value of type int Quad, we need to coerce the recursive type back to Ω:

$$\mathsf{roll}_{(\pi_1(\tau))(\mathsf{int})}(1, 2, 3, 4)$$

Similarly, to create the value $\texttt{Succ}(\texttt{Zero}(\texttt{Q}(1, 2, 3, 4)))$ of type int Square, abbreviating $\texttt{Quad} = \pi_1(\tau)$, $T = \texttt{Quad}(\mathsf{int})$, and $\texttt{Square} = \pi_2(\tau)$, we do the following:

$$\begin{array}{c}\mathsf{roll}_{\texttt{Square}(\mathsf{int})}(\mathsf{inr}_{\mathsf{int}+\texttt{Square}(\texttt{Quad}(\mathsf{int}))}(\mathsf{roll}_{\texttt{Square}(T)} \\ (\mathsf{inl}_{T+\texttt{Square}(\texttt{Quad}(T))}(\mathsf{roll}_T(1, 2, 3, 4)))))\end{array}$$

Unrolling a value of type int Square gives us the *same* value (since unrolling a recursive type is essentially a coercion) of type $\mathsf{int} + (\pi_2(\tau)\ (\pi_1(\tau)\ \mathsf{int}))$. We will revisit these examples when we discuss our new formulation of recursive types in Section 3.

$$
\begin{array}{lll}
\text{(kinds)} & \kappa ::= & \Omega \mid \kappa_1 \Rightarrow \kappa_2 \mid \kappa_1 \circledast \kappa_2 \\
\\
\text{(types)} & \tau ::= & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \mid \boldsymbol{\mu_\kappa(f,g)} \\
 & & \mid t \mid \lambda t : \kappa.\tau \mid \tau_1(\tau_2) \mid \pi_i(\tau) \mid \langle \tau_1, \tau_2 \rangle \\
\\
\text{(expr)} & e ::= & x \mid n \mid op(e_1, e_2) \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \\
 & & \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid e_1(e_2) \mid (e_1, e_2) \mid () \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
 & & \mid \mathbf{roll}_{\mu_\kappa(f,g)}\boldsymbol{(e)} \mid \mathbf{unroll}\boldsymbol{(e)} \mid \mathsf{inl}_{\tau_1+\tau_2}(e_1) \mid \mathsf{inr}_{\tau_1+\tau_2}(e_2) \\
 & & \mid \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inr}(x : \tau_2) \Rightarrow e_2\ \mathsf{end} \\
\\
\text{(oper)} & op ::= & + \mid - \mid \ldots
\end{array}
$$

Fig. 3. A lightweight language using new $\mu : \lambda_{M2}$, changed parts in **bold**

3 A Language Using our Proposed Recursive Types: λ_{M2}

We continue now by modifying λ_{M1} to use our new formulation of recursive types. We'll call this language λ_{M2}. The only differences between λ_{M1} and λ_{M2} involve the handling of recursive types. Figure 3 contains the syntax for this language, with the changed parts in bold. For a full discussion of the formal properties of this language, including static semantics, dynamic semantics, and type-safety theorems, see Appendix B.

3.1 Recursive Types

This language uses our proposed presentation of recursive types. Here μ is a constructor of kind $(\kappa \Rightarrow \kappa) \Rightarrow (\kappa \Rightarrow \Omega) \Rightarrow \Omega$. The relevant portions of the static semantics of this language are as follows:

$$\frac{\Delta \vdash f : \kappa \Rightarrow \kappa \quad \Delta \vdash g : \kappa \Rightarrow \Omega}{\Delta \vdash \mu_\kappa(f,g) : \Omega} \qquad \text{(Well-formed } \mu\text{)}$$

$$\frac{\Delta \vdash \mu_\kappa(f,g) : \Omega \quad \Gamma; \Delta \vdash e : \mu_\kappa(f,g)}{\Gamma; \Delta \vdash \mathsf{unroll}(e) : \mathcal{K}(\kappa, f, g)} \qquad \text{(Unroll)}$$

$$\frac{\Delta \vdash \mu_\kappa(f,g) : \Omega \quad \Gamma; \Delta \vdash e : \mathcal{K}(\kappa, f, g)}{\Gamma; \Delta \vdash \mathsf{roll}_{\mu_\kappa(f,g)}(e) : \mu_\kappa(f,g)} \qquad \text{(Roll)}$$

These last two rules deserve some explanation. Since $f : \kappa \Rightarrow \kappa$, and we have reformulated μ-types to belong to Ω, we cannot unroll types by applying f to the μ-type itself, as we might do in a more standard presentation. Instead we need to analyze the structure of κ to determine how we should unroll the μ-type. This analysis is performed by the macro $\mathcal{K}$ above. We note that the choice of $\mathcal{K}$ does not influence the type-safety properties of the language (see Appendix B). It is conceivable that one might choose an appropriate $\mathcal{K}$ on an application-to-application basis. For this paper, we would like the unroll operator to be as close as possible to the standard one. So, for instance, if $\kappa = \Omega \circledast \Omega$, then we would want the unrolling of $\mu_\kappa(f, g)$ to be:

$$f(\langle \mu_\kappa(f, \lambda k : \kappa.\pi_1(k)), \mu_\kappa(f, \lambda k : \kappa.\pi_2(k))\rangle),$$

which is of kind κ. We would then coerce this, using g, back to Ω. With this in mind, we define $\mathcal{K}$ as follows (assuming that we have the ability to perform a meta-level typecase operation):

$$\begin{array}{c}\mathcal{K}(\kappa, f : \kappa \Rightarrow \kappa, g : \kappa \Rightarrow \Omega) := \\ g(f(\mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, f))),\end{array}$$

where $\mathcal{H}$ is defined as follows:

$$\begin{array}{l}\mathcal{H}(\kappa', \kappa, Q : \kappa \Rightarrow \kappa', f : \kappa \Rightarrow \kappa) := \\ \begin{cases} \mu_\kappa(f, Q) & \text{when } \kappa = \Omega \\ \langle \mathcal{H}(\kappa_1, \kappa, \lambda h : \kappa.(\pi_1(Q\,h)), f), & \\ \quad \mathcal{H}(\kappa_2, \kappa, \lambda h : \kappa.(\pi_2(Q\,h)), f)\rangle & \text{when } \kappa = \kappa_1 \circledast \kappa_2 \\ \lambda g : \kappa_1.\mathcal{H}(\kappa_2, \kappa, \lambda h : \kappa.(Q\,h)(g), f) & \text{when } \kappa = \kappa_1 \Rightarrow \kappa_2. \end{cases}\end{array}$$

The $\mathcal{K}$ macro takes three arguments: the kind κ which we're working over, the fixpoint function $f : \kappa \Rightarrow \kappa$, and the coercion function $g : \kappa \Rightarrow \Omega$. The macro returns the unrolling of $\mu_\kappa(f, g)$. To do this it passes the arguments to the $\mathcal{H}$ macro. The purpose of this macro is to expand the recursive type from something belonging to Ω to something belonging to κ so that f may be applied to it. It takes four parameters: the kind κ' which is the kind of the result, the kind κ which is the kind of the whole expression (necessary for when we create types $\mu_\kappa(f, \dots)$, the function f which is copied verbatim to any μ-types we generate, and a function Q. The Q function works to take a value from κ to κ' — we keep this around to build up the new coercion functions when we create μ-types in the recursive calls.

Revisiting Example 1.1 from the introduction, we specified the type of `tree` to be

$$\mu_{\Omega \circledast \Omega}(f, \lambda t : (\Omega \circledast \Omega).\pi_1(t)),$$

where

$$f = \lambda t : (\Omega \circledast \Omega).\, \langle \mathsf{unit} + (\pi_2(t) * \mathsf{int}), \pi_1(t) * \pi_1(t)\rangle\,.$$

If we were to unroll this type, we would first apply

$$\mathcal{H}(\Omega \circledast \Omega, \Omega \circledast \Omega, \lambda x : \Omega \circledast \Omega.x, f).$$

Substituting once, this leaves:

$$\langle \mathcal{H}(\Omega, \Omega \circledast \Omega, \lambda h : \Omega \circledast \Omega.\pi_1(h), f), \mathcal{H}(\Omega, \Omega \circledast \Omega, \lambda h : \Omega \circledast \Omega.\pi_2(h), f)\rangle\,.$$

Applying $\mathcal{H}(\Omega, \Omega \circledast \Omega, \lambda h : \Omega \circledast \Omega.\pi_1(h), f)$ gives us $\mu_{\Omega \circledast \Omega}(f, \lambda h : \Omega \circledast \Omega.\pi_1(h))$, which (not coincidentally) is the type of `tree` itself. The full unrolling of `tree`, then, is

$$g(f(\langle \mu_{\Omega \circledast \Omega}(f, \lambda h : \Omega \circledast \Omega.\pi_1(h)), \mu_{\Omega \circledast \Omega}(f, \lambda h : \Omega \circledast \Omega.\pi_2(h))\rangle)),$$

which evaluates to

$$\mathsf{unit} + (\mu_{\Omega \circledast \Omega}(f, \lambda x : \Omega \circledast \Omega . \pi_2(x)) * \mathsf{int}),$$

which is precisely what one would expect.

3.2 Examples

We will look again at the example from Okasaki's paper, given in Equation 2.1:

$$\begin{aligned} \texttt{datatype}\ \alpha\,\texttt{Quad} =\ & \texttt{Q of}\ \alpha * \alpha * \alpha * \alpha \\ \texttt{and}\ \alpha\,\texttt{Square} =\ & \texttt{Zero of}\ \alpha \\ & \mid \texttt{Succ of}\ (\alpha\ \texttt{Quad})\,\texttt{Square} \end{aligned}$$

Let $\kappa = (\Omega \Rightarrow \Omega) \circledast (\Omega \Rightarrow \Omega)$. We will type α `Quad` as

$$\texttt{Quad} = \lambda t : \Omega . \mu_\kappa(f, \lambda x : \kappa . (\pi_1(x)\ t)),$$

and we'll type α `Square` as

$$\texttt{Square} = \lambda t : \Omega . \mu_\kappa(f, \lambda x : \kappa . (\pi_2(x)\ t)),$$

where

$$f = \lambda x : \kappa . \, \langle \lambda t : \Omega . t * t * t * t, \ \lambda t : \Omega . t + \pi_2(x)\ (\pi_1(x)\ t) \rangle \, .$$

In order to create a value of type int `Quad`, we would write the following:

$$\mathsf{roll}_{\texttt{Quad(int)}}(1, 2, 3, 4).$$

Similarly, to create the value `Succ(Zero(Q(1, 2, 3, 4)))` of type int `Square`, abbreviating $T = \texttt{Quad(int)}$:

$$\begin{aligned} & \mathsf{roll}_{\texttt{Square(int)}}(\mathsf{inr}_{\texttt{int+Square}(T)} \\ & \quad (\mathsf{roll}_{\texttt{Square}(T)}(\mathsf{inl}_{T+\texttt{Square(Quad}(T))}(\mathsf{roll}_T(1, 2, 3, 4))))) \end{aligned}$$

Unrolling the type int `Square` gives us the type $\mathsf{int} + \tau_2\ (\tau_1\ \mathsf{int})$.

4 Translation from λ_{M1} to λ_{M2}

In this section we will give a translation from λ_{M1} to λ_{M2}. The purpose of giving such a translation is to demonstrate that our new formulation of recursive types is backwards-compatible with the traditional one. To do this, we show that the result of translating a well-typed λ_{M1} term is itself well-typed in λ_{M2}. After inspecting the translation, this should be enough to convince the reader that our new formulation of recursive types is *at least as general* as the traditional one.

We want to embed old-style types $\mu t : \kappa . \tau$ and their associated roll and unroll terms into terms of type $\mu_\kappa(f, g)$.

The translation of μ will preserve kinds. To accomplish this we translate μ types into types having the same kind in which the μ is "pushed" inward. For example, the type

$$\mu t : \Omega \circledast \Omega . \tau$$

would be translated as:

$$\langle \mu_{\Omega \circledast \Omega}(\lambda t : \Omega \circledast \Omega . \tau, \lambda s : \Omega \circledast \Omega . \pi_1(s)), \mu_{\Omega \circledast \Omega}(\lambda t : \Omega \circledast \Omega . \tau, \lambda s : \Omega \circledast \Omega . \pi_2(s)) \rangle \, .$$

4.1 Translation

The translation is as follows:

$$[\![\mu t : \kappa.\tau]\!] = \mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\tau), \tag{4.1}$$

where the $\mathcal{H}$ function is defined in Section 3.1. The translation of other types is trivial; see for example the translation of the lambda-type:

$$[\![\lambda t : \kappa.\tau]\!] = \lambda t : \kappa.\, [\![\tau]\!] \tag{4.2}$$

Type environments stay the same; term environments have their types translated.

$$[\![\Delta]\!] = \Delta \tag{4.3}$$

$$[\![\cdot]\!] = \cdot \tag{4.4}$$

$$[\![\Gamma, x : \tau]\!] = [\![\Gamma]\!], x : [\![\tau]\!] \tag{4.5}$$

Here are the translations for roll and unroll:

$$\frac{\mathsf{p} = \pi_i \mid \cdot \quad \mathsf{t} = \tau'' \mid \cdot \quad \Delta \vdash \tau \equiv (\mathsf{p}(\mu t : \kappa.\tau'))(\mathsf{t}) : \Omega}{[\![\mathsf{roll}_\tau(e)]\!] = \mathsf{roll}_{\mathsf{p}([\![\mu t:\kappa.\tau']\!])([\![\mathsf{t}]\!])}([\![e]\!])} \tag{4.6}$$

$$[\![\mathsf{unroll}(e)]\!] = \mathsf{unroll}([\![e]\!]) \tag{4.7}$$

The following theorem demonstrates the well-typedness of the translation:

Theorem. If $\Gamma; \Delta \vdash e : \tau$, then $[\![\Gamma]\!]; [\![\Delta]\!] \vdash [\![e]\!] : [\![\tau]\!]$.

Proof. In Appendix C.

5 Adding ML-style Datatypes

We now add to our language (in abstract syntax) the ability to define ML-style datatypes, using the treatment in [10]:

$$\begin{array}{rl} \text{(type declarations)} & \delta ::= \mathsf{datatype}\ \beta \\ \text{(datatype bindings)} & \beta ::= (t_1, \ldots, t_n)\ \gamma\ \langle\langle \mathsf{and}\ \beta \rangle\rangle \\ \text{(datatype constructors)} & \gamma ::= \tau\ \langle\langle \mathsf{or}\ \gamma \rangle\rangle, \end{array}$$

where the syntax $\langle\langle X \rangle\rangle$ means X is repeated zero or more times. We note that this is essentially ML syntax, except with identifiers erased (constructors may be accessed by number) and with no restriction on type variables. This presentation is given to simplify the discussion of the type theory.

In a Harper/Stone-style presentation, the type of the i-th datatype in the declaration

$$\begin{array}{l} \mathsf{datatype}\ (t_{11}, \ldots, t_{1n_1})T_1 = \tau_{11}\ \mathsf{or}\ \ldots\ \mathsf{or}\ \tau_{1m_1} \\ \quad \mathsf{and}\ (t_{21}, \ldots, t_{2n_2})T_2 = \tau_{21}\ \mathsf{or}\ \ldots\ \mathsf{or}\ \tau_{2m_2} \\ \quad \mathsf{and}\ \ldots \\ \quad \mathsf{and}\ (t_{p1}, \ldots, t_{pn_p})T_p = \tau_{p1}\ \mathsf{or}\ \ldots\ \mathsf{or}\ \tau_{pm_p} \end{array}$$

is given by

$$\pi_i\Bigg(\mu\lambda(T_1,\ldots,T_p).\\ \quad\Big(\big(\lambda(t_{11},\ldots,t_{1n_1}).(\tau_{11}+\cdots+\tau_{1m_1})\big),\ldots,\big(\lambda(t_{p1},\ldots,t_{pn_p}).(\tau_{p1}+\cdots+\tau_{pm_p})\big)\Big)\Bigg).$$

We'll now show the typing for this datatype using our new presentation (with a slight change in syntax to accommodate n-ary tuples and sums). First we'll define the type σ_i associated with the type parameters of the i-th datatype

$$(t_{i1},\ldots,t_{in_i})T_i = \tau_{i1} \text{ or } \ldots \text{ or } \tau_{im_i}$$

as

$$\sigma_i = \underbrace{\Omega \circledast \cdots \circledast \Omega}_{n_i \text{ times}} \Rightarrow \Omega.$$

Likewise, we define for the i-th datatype clause the sum type $\Sigma_i = \tau_{i1} + \cdots + \tau_{im_i}$. Now we can define the function $F : (\sigma_1 \circledast \cdots \circledast \sigma_p) \Rightarrow (\sigma_1 \circledast \cdots \circledast \sigma_p)$ as:

$$F := \lambda(T_1 : \sigma_1,\ldots,T_p : \sigma_p).\,\langle\Sigma_1,\ldots,\Sigma_n\rangle\,.$$

Now that we've defined F, we're in a position to type the entire datatype declaration T_i with the type:

$$\lambda(t_{i1} : \Omega,\ldots,t_{in_i} : \Omega).\mu_{\sigma_1\circledast\cdots\circledast\sigma_p}(F, \lambda t : \sigma_1 \circledast \cdots \circledast \sigma_p.(\#i(t)(t_{i1},\ldots,t_{in_i}))).$$

6 Intensional Type Analysis

6.1 Introduction

Supporting type analysis type-safely has been an active area of research in past years, since Harper and Morrisett [1] introduced the *intensional* type analysis (ITA) framework — that is, analysis of the structure of types. Intensional type analysis was novel in that it allows for the inspection of types in a type-safe manner. The Harper/Morrisett framework, however, only supports primitive recursion over the *monotypes* of their language λ_i^{ML}, cannot express general recursion at all, and cannot support analysis of types with binding structure (such as polymorphic or existential types).

Existing work in this area [6, 9, 11] either ignores or does not fully support analysis of recursive types. Trifonov et al. [6] present a framework which ostensibly supports *fully reflexive* ITA — by "fully reflexive", meaning that type-analyzing operations are applicable to the type of any runtime value in the language. Their framework's support of recursive types is severely limited, however. In this section we build upon this result and present a lightweight language which fully supports the type-level analysis of recursive types.

$$\begin{array}{lll}
\text{(exp)} & e ::= & x \mid n \mid op(e_1, e_2) \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \\
& & \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid e_1(e_2) \mid (e_1, e_2) \mid () \mid \#i(e) \\
& & \mid \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{or}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2 \\
& & \mid \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(e_i) \mid \mathsf{roll}_\tau(e) \mid \mathsf{unroll}(e) \\
\text{(oper)} & op ::= & + \mid - \mid \ldots
\end{array}$$

Fig. 4. Syntax of the term language λ_i^R

6.2 CiC

To typecheck this term language, we will use the calculus of inductive constructions (CiC) [14], which is implemented in the Coq proof assistant [15]. CiC is an extension of the calculus of constructions (CC) [16], which is a higher-order typed lambda calculus. The syntax of CC is as follows:

$$\boxed{A, B ::= \mathsf{Set} \mid \mathsf{Type} \mid X \mid \lambda X : A.B \mid AB \mid \Pi X : A.B}$$

The *lambda* term denotes a function abstraction, and the Π term denotes a dependent product type. When X does not occur in B, the term $\Pi X : A.B$ is usually abbreviated $A \rightarrow B$.

CiC extends the calculus of constructions with inductive definitions. An inductive definition is usually written in a syntax similar to that of an ML datatype definition; the following is a snippet of Coq code defining the set of natural numbers:

$$\mathsf{Inductive}\ \mathtt{Nat} : \mathsf{Set} := \mathtt{zero} : \mathtt{Nat} \mid \mathtt{succ} : \mathtt{Nat} \rightarrow \mathtt{Nat}$$

Inductive definitions in CiC may be parameterized, and the calculus provides elimination constructs for inductive definitions, which combine case analysis with a fixpoint operation. To maintain the soundness of the calculus, restrictions are placed on the constructors of inductive definitions, to ensure that the inductive type being created occurs only *positively* in the definition of the constructor. A full discussion of inductive types is beyond the scope of this paper. Elimination on inductive definitions generalizes the Typerec operator used in previous ITA papers [1, 6, 9].

6.3 Term Language

We are now in a position to present our term language. See Figure 4 for the syntax of λ_i^R. To simplify our analysis we only consider constants representing natural numbers (n). We present only pairs, not tuples (tuples may be defined as a "syntactic sugar" on pairs). The term language also lacks function polymorphism and existential types, as ITA on these types has been dealt with in previous papers [11, 12].

Static Semantics We'll be typechecking λ_i^R using CIC. We define the type constructors of the term language as constructors of an inductive kind Ω as follows:

$$\begin{array}{lll}
\mathsf{Inductive}\ \Omega : \mathsf{Set} ::= & \mathsf{int} & : \Omega \\
& \mid \mathsf{unit} & : \Omega \\
& \mid \twoheadrightarrow & : \Omega \to \Omega \to \Omega \\
& \mid \mathsf{pair} & : \Omega \to \Omega \to \Omega \\
& \mid \mathsf{sumty} & : \Omega \to \Omega \to \Omega \\
& \mid \boldsymbol{\mu} & : \Pi k : \mathsf{Set}.(k \to k) \to (k \to \Omega) \to \Omega
\end{array}$$

In other words, all terms in λ_i^R have a type τ which belongs to Ω. To improve readability, we define the following syntactic sugars:

$$\begin{aligned}
A_1 \to A_2 &\equiv \twoheadrightarrow A_1\, A_2 \\
A_1 * A_2 &\equiv \mathsf{pair}\, A_1\, A_2 \\
A_1 + A_2 &\equiv \mathsf{sumty}\, A_1\, A_2 \\
\mu_\kappa(f, g) &\equiv \boldsymbol{\mu}[\kappa]\, f\, g
\end{aligned}$$

In order to be able to unroll recursive types, we need to restrict the kinds upon which a recursive type may operate. Hence we define the inductive kind (using Coq syntax[1]):

$$\begin{array}{lll}
\mathsf{Inductive}\ \mathtt{Kind} : \mathsf{Set} ::= & \omega & : \mathtt{Kind} \\
& \mid \mathtt{To} & : \mathtt{Kind} \to \mathtt{Kind} \to \mathtt{Kind} \\
& \mid \mathtt{Pair} & : \mathtt{Kind} \to \mathtt{Kind} \to \mathtt{Kind}
\end{array}$$

To simplify the presentation, here we have only defined pair kinds; however, this may be extended to tuple kinds without much trouble. Given this inductive definition, we would like to coerce elements of this kind back to their representations:

```
Fixpoint toSet[$K$ : Kind] : Set :=
    Cases $K$ of
        $\omega \Rightarrow \Omega$
        | (To $k_1$ $k_2$) $\Rightarrow$ (toSet $k_1$) $\to$ (toSet $k_2$)
        | (Pair $k_1$ $k_2$) $\Rightarrow$ (toSet $k_1$) $*$ (toSet $k_2$)
    end.
```

Here the arrow and star in the right-hand side of the case clauses are the type-level arrow and cartesian product types (where cartesian product is also defined as an inductive type). We define the following syntactic sugars for these kinds:

$$\begin{aligned}
A_1 \Rightarrow A_2 &\equiv \mathtt{To}\, A_1\, A_2 \\
A_1 \circledast A_2 &\equiv \mathtt{Pair}\, A_1\, A_2
\end{aligned}$$

For a full discussion of the formal properties of λ_i^R, including static semantics, dynamic semantics, and type-safety theorems, please see Appendix D.

[1] Coq includes syntactic sugars for fixpoint operations and inductive eliminations. Lambda abstractions are written as $[X : A]B$, and Π-terms are written as $(X : A)B$.

6.4 An Application: CPS Conversion

In this section we will show how to perform CPS conversion on λ_i^R using intensional type analysis. CPS conversion transforms all unconditional control transfers, including function invocation and return, to function calls and gives explicit names to all intermediate computations. To perform CPS conversion, we will need a target language which we will call $\lambda_{\mathcal{K}}$, with syntax:

$$\begin{array}{lrl}
\text{(val)} & v ::= & x \mid n \mid () \mid (v_1, v_2) \mid \mathsf{roll}_\tau(v) \\
& & \mid \mathsf{fix}\ x'[X_1 : A_1, \ldots, X_n : A_n](x : A).e \mid \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(v) \\
\text{(exp)} & e ::= & v[A_1, \ldots, A_n](v') \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \#i(v)\ \mathsf{in}\ e \\
& & \mid \mathsf{let}\ x = op(v_1, v_2)\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{unroll}(v)\ \mathsf{in}\ e \\
& & \mid \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
& & \mid \mathsf{let}\ x = \mathsf{case}_{\tau_1+\tau_2} v\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{or}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2\ \mathsf{in}\ e \\
\text{(oper)} & op ::= & + \mid - \mid \ldots
\end{array}$$

Expressions in $\lambda_{\mathcal{K}}$ consist of a series of let bindings followed by a function application or a conditional branch. There is only one abstraction mechanism, fix, which combines type and value abstraction.

The $\lambda_{\mathcal{K}}$ language uses the same type language as λ_i^R. The ground types for $\lambda_{\mathcal{K}}$ all have kind $\Omega_{\mathcal{K}}$, which, as before, is an inductive kind defined in CIC. The $\Omega_{\mathcal{K}}$ language has all the constructors of Ω. Functions in CPS do not return values, so in $\Omega_{\mathcal{K}}$ we redefine the $\twoheadrightarrow$ constructor to have kind:

$$\twoheadrightarrow : \Omega_{\mathcal{K}} \to \Omega_{\mathcal{K}}.$$

We use the more conventional syntax $A \to \bot$ in place of $\twoheadrightarrow A$.

Typed CPS conversion involves translating both types and computation terms. To translate types, we require a function $\mathsf{K}_{\mathrm{typ}} : \Omega \to \Omega_{\mathcal{K}}$, and to translate terms we require a meta-level function $\mathsf{K}_{\mathrm{exp}}$. Since we have used CIC as a general framework using inductive definitions, we can write $\mathsf{K}_{\mathrm{typ}}$ as a type-level function of kind $\Omega \to \Omega_{\mathcal{K}}$. This allows us to show that $\mathsf{K}_{\mathrm{typ}}\ [\![[A/X]B]\!]$ and $[A/X](\mathsf{K}_{\mathrm{typ}}\ [\![B]\!])$ are equivalent. This would prove very difficult if $\mathsf{K}_{\mathrm{typ}}$ were defined at the meta-level.

We can define $\mathsf{K}_{\mathrm{typ}}$ then as follows (using pattern-matching syntax):

$$\begin{array}{lcl}
\mathsf{K}_{\mathrm{typ}}(\mathsf{int}) & = & \mathsf{int} \\
\mathsf{K}_{\mathrm{typ}}(\mathsf{unit}) & = & \mathsf{unit} \\
\mathsf{K}_{\mathrm{typ}}(\twoheadrightarrow\ t_1\ t_2) & = & (\mathsf{K}_{\mathrm{typ}}(t_1) * \mathsf{K}_c(t_2)) \to \bot \\
\mathsf{K}_{\mathrm{typ}}(\mathsf{pair}\ t_1\ t_2) & = & \mathsf{pair}\ \mathsf{K}_{\mathrm{typ}}(t_1)\ \mathsf{K}_{\mathrm{typ}}(t_2) \\
\mathsf{K}_{\mathrm{typ}}(\mathsf{sumty}\ t_1\ t_2) & = & \mathsf{sumty}\ \mathsf{K}_{\mathrm{typ}}(t_1)\ \mathsf{K}_{\mathrm{typ}}(t_2) \\
\mathsf{K}_{\mathrm{typ}}(\boldsymbol{\mu}[k]\ f\ g) & = & \boldsymbol{\mu}[k]\ f\ \lambda x : k.(\mathsf{K}_{\mathrm{typ}}(g\ x)) \\
\mathsf{K}_c & = & \lambda t : \Omega.\mathsf{K}_{\mathrm{typ}}(t) \to \bot
\end{array}$$

The complete CPS-conversion algorithm is given in Section D.5. The handling of the μ constructor in this example deserves some illumination. We note firstly that when we translate a roll term from λ_i^R to $\lambda_{\mathcal{K}}$, that the f function is unchanged in $\mu_\kappa(f, g)$. This means that in the static semantics for $\lambda_{\mathcal{K}}$, the f function is still limited to kinds matching `toSet`(k), which work over Ω in the base case. What happens, then, when we unroll such a term? The $\mathsf{K}_{\mathrm{typ}}$ function composes the coercion function g with itself, so when we unroll $\mu_\kappa(f, g')$, we obtain

$$\mathsf{K}_{\mathrm{typ}}(g(f(H(\dots)))),$$

and any instances of μ in the unrolled type have their coercion functions likewise altered.

7 Related Work and Conclusions

Harper and Morrisett [1] 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 and it does not cover quantified types and recursive types.

Crary and Weirich [9] proposed the use of deBruijn indices (i.e. natural numbers) to represent quantifier-bound variables. To analyze recursive 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 works well on recursive types of kind Ω but would be difficult, if not impossible, to extend to higher kinds. Papers by Trifonov et al. [6] and Weirich [17] exhibit the same problem.

The original motivation for this paper was to develop a formulation of recursive types which fit into a pre-existing framework which made pervasive use of inductive definitions. We believe, however, that the solution we have presented in this paper is applicable in a far more general context. Our goal for this paper is to put recursive types on a more rigorous, or at least more consistent, footing. The technique that we have proposed is simple, general, and extensible. It is consistent enough with standard approaches to type theory that we could, for instance, replace the recursive types in Harper's SML type theory paper [10] with ours with minimal effort. We believe that our paper is the first one to successfully attack the problem of fitting mutually recursive types in the context of intensional type analysis.

7.1 Future work

Whereas other approaches to higher-order intensional type analysis [6, 9, 17] support *runtime* intensional type analysis, (also known as *polytypic programming*), our treatment here, because of space restrictions, only considers type-level analysis. We are confident that we will be able to extend λ_i^R to support a term-level typecase operator with little difficulty.

Acknowledgements. Thanks to Hai Fang, Andrew McCreight, and Stefan Monnier for their helpful suggestions.

Bibliography

[1] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. POPL '95: 22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, USA, 1995.

[2] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proc. POPL '02: 29th ACM Symposium on Principles of Programming Languages*, pages 217–232, Portland, OR, USA, January 2002.

[3] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. Twenty-Sixth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

[4] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[5] Paul Hudak, Simon Peyton Jones, Philip Wadler, and *et al.* Report on the programming language Haskell, a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 21 (5), May 1992.

[6] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proc. ICFP '00: ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, September 2000.

[7] Bratin Saha, Valery Trifonov, and Zhong Shao. Intensional analysis of quantified types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2002. To appear.

[8] George C. Necula. Proof-carrying code. In *Proc. POPL '97: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[9] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proc. ICFP '99: ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, 1999.

[10] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, June 1997.

[11] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis in type erasure semantics. Technical Report YALEU/DCS/TR-1197, Dept. of Computer Science, Yale University, New Haven, CT, USA, June 2000.

[12] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis. Technical Report YALEU/DCS/TR-1194, Dept. of Computer Science, Yale University, New Haven, CT, USA, March 2000.

[13] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *Proc. ICFP '99: ACM SIGPLAN International Conference on Functional Programming*, pages 28–35, 1999.

[14] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[15] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.1.* INRIA-Rocquencourt-CNRS-ENS Lyon, October 2001.

[16] Thierry Coquand and Gerard Huet. The calculus of constructions. In *Information and Computation*, volume 76, pages 95–120, 1988.

[17] Stephanie Weirich. Higher-order intensional type analysis. In *European Symposium on Programming*, pages 98–114, 2002.

[18] Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Accepted for publication

in the Archive for Mathematical Logic. Available from `http://www.tcs.informatik.uni-muenchen.de/~matthes/abstracts.html.`, 2002.

[19] Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, L'Université Paris 7, May 1994.

A Formal Properties of λ_{M1}

A.1 Static semantics

Environments We define two environments, Δ and Γ. The Δ environment is a *type environment* mapping type variables to their kinds, and Γ is a *term environment* mapping term variables to their types. Typing a term e with a type τ is denoted as $\Gamma; \Delta \vdash e : \tau$. The static semantics for these environments are as follows:

$$\vdash \Omega\ \mathit{kind} \tag{A.1}$$

$$\frac{\vdash \kappa_1\ \mathit{kind} \quad \vdash \kappa_2\ \mathit{kind}}{\vdash \kappa_1 \Rightarrow \kappa_2\ \mathit{kind}} \tag{A.2}$$

$$\frac{\vdash \kappa_1\ \mathit{kind} \quad \vdash \kappa_2\ \mathit{kind}}{\vdash \kappa_1 \circledast \kappa_2\ \mathit{kind}} \tag{A.3}$$

$$\vdash \cdot\ \mathit{type\ env} \tag{A.4}$$

$$\frac{\vdash \Delta\ \mathit{type\ env} \quad \vdash \kappa\ \mathit{kind}}{\vdash \Delta, t : \kappa\ \mathit{type\ env}} \tag{A.5}$$

$$\Delta \vdash \cdot\ \mathit{term\ env} \tag{A.6}$$

$$\frac{\Delta \vdash \Gamma\ \mathit{term\ env} \quad \Delta \vdash \tau : \Omega}{\Delta \vdash \Gamma, x : \tau\ \mathit{term\ env}} \tag{A.7}$$

Type formation $\boxed{\Delta \vdash \tau : \kappa}$

$$\frac{t \in \Delta}{\Delta \vdash t : \Delta(t)} \tag{A.8}$$

$$\frac{}{\Delta \vdash \mathsf{int} : \Omega} \tag{A.9}$$

$$\frac{}{\Delta \vdash \mathsf{unit} : \Omega} \tag{A.10}$$

$$\frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \Omega} \tag{A.11}$$

$$\frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 + \tau_2 : \Omega} \tag{A.12}$$

$$\frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 * \tau_2 : \Omega} \tag{A.13}$$

$$\frac{\Delta, t : \kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda t{:}\kappa_1.\tau : \kappa_1 \Rightarrow \kappa_2} \tag{A.14}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \Rightarrow \kappa_2 \quad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2} \tag{A.15}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \quad \Delta \vdash \tau_2 : \kappa_2}{\Delta \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \circledast \kappa_2} \tag{A.16}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_i(\tau) : \kappa_i} \tag{A.17}$$

$$\frac{\Delta, t : \kappa \vdash \tau : \kappa}{\Delta \vdash \mu t : \kappa.\tau : \kappa} \tag{A.18}$$

$$\frac{\Delta \vdash \Gamma(x) : \Omega \quad (\forall x \in \mathrm{Dom}(\Gamma))}{\Delta \vdash \Gamma} \tag{A.19}$$

Type equality $\boxed{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa}$

Rule A.20: reflexivity.

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau \equiv \tau : \kappa} \tag{A.20}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa}{\Delta \vdash \tau_2 \equiv \tau_1 : \kappa} \tag{A.21}$$

Rule A.21: symmetry.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa \quad \Delta \vdash \tau_2 \equiv \tau_3 : \kappa}{\Delta \vdash \tau_1 \equiv \tau_3 : \kappa} \tag{A.22}$$

Rule A.22: transitivity.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Delta \vdash \tau_2 \equiv \tau_2' : \Omega}{\Delta \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \Omega} \tag{A.23}$$

We note here that the function type is only defined for types of kind Ω; see rule A.11.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Delta \vdash \tau_2 \equiv \tau_2' : \Omega}{\Delta \vdash \tau_1 + \tau_2 \equiv \tau_1' + \tau_2' : \Omega} \tag{A.24}$$

Note that the sum type is only defined for types of kind Ω; see rule A.12.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Delta \vdash \tau_2 \equiv \tau_2' : \Omega}{\Delta \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2' : \Omega} \tag{A.25}$$

Note, again, that the pair type is only defined for types of kind Ω; see rule A.13.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \kappa_1 \quad \Delta \vdash \tau_2 \equiv \tau_2' : \kappa_2}{\Delta \vdash \langle \tau_1, \tau_2 \rangle \equiv \langle \tau_1', \tau_2' \rangle : \kappa_1 \circledast \kappa_2} \tag{A.26}$$

$$\frac{\Delta, t : \kappa \vdash \tau_1 \equiv \tau_2[s := t] : \kappa \quad t \notin FV(\tau_2)}{\Delta \vdash \mu t : \kappa.\tau_1 \equiv \mu s : \kappa.\tau_2 : \kappa} \tag{A.27}$$

$$\frac{\Delta \vdash \tau_1 \equiv \lambda t : \kappa.\tau : \kappa \Rightarrow \kappa' \quad \Delta \vdash \tau_2 : \kappa}{\Delta \vdash \tau_1 \tau_2 \equiv \tau_1[t := \tau_2] : \kappa'} \tag{A.28}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \kappa_1 \Rightarrow \kappa_2 \quad \Delta \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Delta \vdash \tau_1(\tau_2) \equiv \tau_1'(\tau_2') : \kappa_2} \tag{A.29}$$

$$\frac{\Delta, t : \kappa \vdash \tau \equiv \tau'[s := t] : \kappa' \quad t \notin FV(\tau')}{\Delta \vdash \lambda t : \kappa.\tau \equiv \lambda s : \kappa.\tau' : \kappa \Rightarrow \kappa'} \tag{A.30}$$

$$\frac{\Delta \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_i(\langle \tau_1, \tau_2 \rangle) \equiv \tau_i : \kappa_i} \tag{A.31}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_i(\tau_1) \equiv \pi_i(\tau_2) : \kappa_i} \tag{A.32}$$

Term formation

$$\boxed{\Gamma; \Delta \vdash e : \tau}$$

$$\frac{x \in \Gamma}{\Gamma; \Delta \vdash x : \Gamma(x)} \tag{A.33}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 \quad \Gamma; \Delta \vdash e : \tau_1}{\Gamma; \Delta \vdash e : \tau_2} \tag{A.34}$$

$$\frac{}{\Gamma; \Delta \vdash n : \mathsf{int}} \tag{A.35}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \mathsf{int} \quad \Gamma; \Delta \vdash e_2 : \mathsf{int}}{\Gamma; \Delta \vdash op(e_1, e_2) : \mathsf{int}} \tag{A.36}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1;\ \Delta \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} :\ \tau_1 \to \tau_2} \tag{A.37}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta \vdash e_2 : \tau_1}{\Gamma; \Delta \vdash e_1(e_2) : \tau_2} \tag{A.38}$$

$$\frac{}{\Gamma; \Delta \vdash () : \mathsf{unit}} \tag{A.39}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \quad \Gamma; \Delta \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash (e_1, e_2) : \tau_1 * \tau_2} \tag{A.40}$$

$$\frac{\Gamma; \Delta \vdash e : \mathsf{int} \quad \Gamma; \Delta \vdash e_1 : \tau \quad \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} : \tau} \tag{A.41}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1 * \tau_2}{\Gamma; \Delta \vdash \mathsf{fst}(e) : \tau_1} \tag{A.42}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1 * \tau_2}{\Gamma; \Delta \vdash \mathsf{snd}(e) : \tau_2} \tag{A.43}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1}{\Gamma; \Delta \vdash \mathsf{inl}_{\tau_1 + \tau_2}(e) : \tau_1 + \tau_2} \tag{A.44}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathsf{inr}_{\tau_1 + \tau_2}(e) : \tau_1 + \tau_2} \tag{A.45}$$

$$\frac{\begin{array}{c}\Gamma, x_2 : \tau_2; \Delta \vdash e_2 : \tau \\ \Gamma, x_1 : \tau_1; \Delta \vdash e_1 : \tau \\ \Gamma; \Delta \vdash e : \tau_1 + \tau_2\end{array}}{\begin{array}{c}\Gamma; \Delta \vdash \mathsf{case}_{\tau_1 + \tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x_1 : \tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x_2 : \tau_2) \Rightarrow e_2\ \mathsf{end} : \tau\end{array}} \tag{A.46}$$

$$\frac{\begin{array}{c}\mathsf{p} = \pi_i \mid \cdot \\ \mathsf{s} = \tau'' \mid \cdot \\ \Delta \vdash \tau \equiv (\mathsf{p}(\mu t : \kappa.\tau'))(\mathsf{s}) : \Omega \\ \Gamma; \Delta \vdash e : \tau\end{array}}{\Gamma; \Delta \vdash \mathsf{unroll}(e) : (\mathsf{p}(\tau'[t := \mu t : \kappa.\tau']))(\mathsf{s})} \tag{A.47}$$

$$\frac{\begin{array}{c}\mathsf{p} = \pi_i \mid \cdot \\ \mathsf{s} = \tau'' \mid \cdot \\ \Delta \vdash \tau \equiv (\mathsf{p}(\mu t : \kappa.\tau'))(\mathsf{s}) : \Omega \\ \Gamma; \Delta \vdash e : (\mathsf{p}(\tau'[t := \mu t : \kappa.\tau']))(\mathsf{s})\end{array}}{\Gamma; \Delta \vdash \mathsf{roll}_{\tau}(e)} \tag{A.48}$$

A.2 Dynamic semantics

The values of the language are as follows:

$$\begin{aligned}(\text{values})\ v ::= {} & n \mid \mathsf{fun}\ f(x:\tau_1):\tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid \mathsf{inl}_{\tau_1+\tau_2}(v) \mid \mathsf{inr}_{\tau_1+\tau_2}(v) \\ & \mid () \mid (v_1, v_2) \mid \mathsf{roll}_{\tau}(v)\end{aligned}$$

Primitive evaluation rules

$$\frac{}{\mathrm{op}(v_1, v_2) \hookrightarrow v_1\ \mathrm{op}\ v_2} \qquad \text{(A.49)}$$

$$\frac{v_1 = \mathsf{fun}\ f(x:\tau_1):\tau_2\ \mathsf{is}\ e\ \mathsf{end}}{v_1(v_2) \hookrightarrow e[f := v_1, x := v_2]} \qquad \text{(A.50)}$$

$$\frac{}{\mathsf{fst}(v_1, v_2) \hookrightarrow v_1} \qquad \text{(A.51)}$$

$$\frac{}{\mathsf{snd}(v_1, v_2) \hookrightarrow v_2} \qquad \text{(A.52)}$$

$$\frac{v = \mathsf{inl}_{\tau_1+\tau_2}(v')}{\begin{array}{c}\mathsf{case}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ \mathsf{inl}(x:\tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x:\tau_2) \Rightarrow e_2\ \mathsf{end} \\ \hookrightarrow e_1[x := v']\end{array}} \qquad \text{(A.53)}$$

$$\frac{v = \mathsf{inr}_{\tau_1+\tau_2}(v')}{\begin{array}{c}\mathsf{case}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ \mathsf{inl}(x:\tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x:\tau_2) \Rightarrow e_2\ \mathsf{end} \\ \hookrightarrow e_2[x := v']\end{array}} \qquad \text{(A.54)}$$

$$\frac{}{\mathsf{unroll}(\mathsf{roll}_{\tau}(v)) \hookrightarrow v} \qquad \text{(A.55)}$$

$$\frac{v \mathrel{/}= 0}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_1} \qquad \text{(A.56)}$$

$$\frac{v = 0}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_2} \qquad \text{(A.57)}$$

Search rules

$$\frac{e_1 \hookrightarrow e_1'}{\mathrm{op}(e_1, e_2) \hookrightarrow \mathrm{op}(e_1', e_2)} \qquad \text{(A.58)}$$

$$\frac{e_2 \hookrightarrow e_2'}{\mathrm{op}(v, e_2) \hookrightarrow \mathrm{op}(v, e_2')} \qquad \text{(A.59)}$$

$$\frac{e_1 \hookrightarrow e_1'}{e_1(e_2) \hookrightarrow e_1'(e_2)} \qquad \text{(A.60)}$$

$$\frac{e_2 \hookrightarrow e_2'}{v(e_2) \hookrightarrow v(e_2')} \qquad \text{(A.61)}$$

$$\frac{e_1 \hookrightarrow e_1'}{(e_1, e_2) \hookrightarrow (e_1', e_2)} \qquad \text{(A.62)}$$

$$\frac{e_2 \hookrightarrow e_2'}{(v_1, e_2) \hookrightarrow (v_1, e_2')} \qquad \text{(A.63)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{fst}(e) \hookrightarrow \mathsf{fst}(e')} \qquad \text{(A.64)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{snd}(e) \hookrightarrow \mathsf{snd}(e')} \qquad \text{(A.65)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{inl}_{\tau_1+\tau_2}(e) \hookrightarrow \mathsf{inl}_{\tau_1+\tau_2}(e')} \qquad \text{(A.66)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{inr}_{\tau_1+\tau_2}(e) \hookrightarrow \mathsf{inr}_{\tau_1+\tau_2}(e')} \qquad \text{(A.67)}$$

$$\frac{e \hookrightarrow e'}{\begin{array}{c}\mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x:\tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x:\tau_2) \Rightarrow e_2\ \mathsf{end} \\ \hookrightarrow \mathsf{case}_{\tau_1+\tau_2}\ e'\ \mathsf{of}\ \mathsf{inl}(x:\tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x:\tau_2) \Rightarrow e_2\ \mathsf{end}\end{array}} \qquad \text{(A.68)}$$

$$\frac{e \hookrightarrow e'}{\begin{array}{c}\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \\ \hookrightarrow \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}\end{array}} \qquad \text{(A.69)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{unroll}(e) \hookrightarrow \mathsf{unroll}(e')} \qquad \text{(A.70)}$$

$$\frac{e \hookrightarrow e'}{\mathsf{roll}_{\tau}(e) \hookrightarrow \mathsf{roll}_{\tau}(e')} \qquad \text{(A.71)}$$

A.3 Soundness and safety

Properties of static semantics

Proposition 1 (Strong normalization). *Type reduction for the language given above is strongly normalizing.*

Proof (sketch). Rules A.20 through A.32 describe a simply-typed lambda calculus with pair types and constants. Proofs of strong normalization for $\lambda \rightarrow$ are familiar in the literature (see [18], for instance.)

Corollary 2 (Decidability). *Type-checking the language given above is decidable.*

Proof. Judgments for the formations of kinds (A.1–A.3), type environments (A.4–A.5), term environments (A.6–A.7), and type formation are all syntax-directed and hence decidable.

Type equivalence, however, is not syntax-directed. Since type reductions are strongly normalizing (Proposition 1), we can determine whether $\tau_1 \equiv \tau_2$ by reducing both τ_1 and τ_2 to normal form, then testing whether the normal forms are syntactically congruent.

Term formation is syntax-directed, excepting the type-equivalence rule A.34. However, if the type-checker always reduces types to normal form, then rule A.34 can be omitted.

Soundness

Lemma 3 (Substitution).

1. *If* $\Gamma; \Delta \vdash e' : \tau'$ *and* $\Gamma, x : \tau'; \Delta \vdash e : \tau$*, then* $\Gamma; \Delta \vdash e[x := e'] : \tau$.
2. *If* $\Delta \vdash \tau' : \kappa$ *and* $\Gamma; \Delta, t : \kappa \vdash e : \tau$*, then* $\Gamma[t := \tau']; \Delta \vdash e[t := \tau'] : \tau[t := \tau']$.

Proof.

1. By induction on the derivation of $\Gamma, x : \tau'; \Delta \vdash e : \tau$.
2. By induction on the derivation of $\Gamma; \Delta, t : \kappa \vdash e : \tau$.

Lemma 4 (Type Inversion).

1. *If* $\Gamma; \Delta \vdash x : \tau$*, then* $\Delta \vdash \Gamma(x) \equiv \tau : \Omega$.
2. *If* $\Gamma; \Delta \vdash n : \tau$*, then* $\Delta \vdash \tau \equiv \mathit{int} : \Omega$.
3. *If* $\Gamma; \Delta \vdash \mathit{op}(e_1, e_2) : \tau$*, then* $\Delta \vdash \tau \equiv \mathit{int} : \Omega$, $\Gamma; \Delta \vdash e_1 : \mathit{int}$*, and* $\Gamma; \Delta \vdash e_2 : \mathit{int}$.
4. *If* $\Gamma; \Delta \vdash \mathit{fun}\, f(x : \tau_1) : \tau_2\ \mathit{is}\ e\ \mathit{end} : \tau$*, then* $\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1; \Delta \vdash e : \tau_2$ *and* $\Delta \vdash \tau \equiv \tau_1 \rightarrow \tau_2 : \Omega$.
5. *If* $\Gamma; \Delta \vdash e_1(e_2) : \tau$*, then there exists* τ_2 *such that* $\Gamma; \Delta \vdash e_1 : \tau_2 \rightarrow \tau$ *and* $\Gamma; \Delta \vdash e_2 : \tau_2$.
6. *If* $\Gamma; \Delta \vdash (e_1, e_2) : \tau$*, then there exist* τ_1 *and* τ_2 *such that* $\Delta \vdash \tau \equiv \tau_1 * \tau_2 : \Omega$, $\Gamma; \Delta \vdash e_1 : \tau_1$*, and* $\Gamma; \Delta \vdash e_2 : \tau_2$.
7. *If* $\Gamma; \Delta \vdash () : \tau$*, then* $\Delta \vdash \tau \equiv \mathit{unit} : \Omega$.
8. *If* $\Gamma; \Delta \vdash \mathit{fst}(e) : \tau$*, then there exists* τ_2 *such that* $\Gamma; \Delta \vdash e : \tau * \tau_2$*, and similarly for snd.*
9. *If* $\Gamma; \Delta \vdash \mathit{inl}_{\tau_1+\tau_2}(e_1) : \tau$*, then* $\Delta \vdash \tau \equiv \tau_1 + \tau_2 : \Omega$ *and* $\Gamma; \Delta \vdash e_1 : \tau_1$*. The inr case is similar.*
10. *If* $\Gamma; \Delta \vdash \mathit{case}_{\tau_1+\tau_2}\ e\ \mathit{of}\ \mathit{inl}(x : \tau_1) \Rightarrow e_1\ \mathit{orelse}\ \mathit{inr}(x : \tau_2) \Rightarrow e_2\ \mathit{end} : \tau$*, then* $\Gamma; \Delta \vdash e : \tau_1 + \tau_2$, $\Gamma, x_1 : \tau_1; \Delta \vdash e_1 : \tau$*, and* $\Gamma, x_2 : \tau_2; \Delta \vdash e_2 : \tau$.
11. *If* $\Gamma; \Delta \vdash \mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{fi} : \tau$*, then* $\Gamma; \Delta \vdash e : \mathit{int}$, $\Gamma; \Delta \vdash e_1 : \tau$*, and* $\Gamma; \Delta \vdash e_2 : \tau$.
12. *Let* $p = \cdot \mid \pi_i$*, and let* $t = \cdot \mid \tau'''$*. If* $\Gamma; \Delta \vdash \mathit{roll}_\tau(e) : \tau'$*, then* $\Delta \vdash \tau \equiv \tau' \equiv p(\mu t : \kappa.\tau'')(t) : \Omega$ *and* $\Gamma; \Delta \vdash e : p(\tau''[t := \mu t : \kappa.\tau''])(t)$.
13. *Let* $p = \cdot \mid \pi_i$*, and let* $t = \cdot \mid \tau''$*. If* $\Gamma; \Delta \vdash \mathit{unroll}(e) : \tau$*, then* $\Delta \vdash \tau \equiv (p(\tau'[t := \mu t : \kappa.\tau']))(t)$ *and* $\Gamma; \Delta \vdash e : (p(\mu t : \kappa.\tau'))(t)$.

Proof (sketch). The proof relies on the fact that every typing derivation $\Gamma; \Delta \vdash e : \tau$ is the result of a unique typing rule, apart from the type-equality rule A.34. The type-equality rule is dealt with by noting that since it has only one premise, each typing derivation ending with $\Gamma; \Delta \vdash e : \tau$ is a finite chain of zero or more type-equality judgments, with a necessarily unique type judgment at the top.

Theorem 5 (Preservation). *If* $\Gamma; \Delta \vdash e : \tau$ *and* $e \hookrightarrow e'$*, then* $\Gamma; \Delta \vdash e' : \tau$.

Proof. The proof is by induction on the rules defining one-step evaluation.

(Rule A.49) Here $e = \mathrm{op}(v_1, v_2)$ and $e' = v_1 \;\mathrm{op}\; v_2$. By type inversion rule 3, $\Delta \vdash \tau \equiv \mathsf{int} : \Omega$, $\Gamma; \Delta \vdash v_1 : \mathsf{int}$, and $\Gamma; \Delta \vdash v_2 : \mathsf{int}$. The meta-level integer operation "op" returns an integer, so $\Gamma; \Delta \vdash e' : \mathsf{int}$.

(Rule A.50) Here $e = v_1(v_2)$, where $v_1 = \mathsf{fun}\, f(x : \tau_1) : \tau_2 \;\mathsf{is}\; e_1 \;\mathsf{end}$. From syntax and type inversion rule 5, we know that $\Gamma; \Delta \vdash e : \tau_2$, and that $\Gamma; \Delta \vdash v_1 : \tau_1 \to \tau_2$. Using inversion again (rule 4), we have that $\Gamma, f : \tau_1 \to \tau_2, x : \tau_1; \Delta \vdash e_1 : \tau_2$. By twice applying the substitution lemma (3), we have that $\Gamma; \Delta \vdash e_1[f := v_1, x := v_2] : \tau_2$, which is what is required.

(Rule A.51) Here $e = \mathsf{fst}(v_1, v_2)$. By type inversion rule 8, we have $\Gamma; \Delta \vdash (v_1, v_2) : \tau_1 * \tau_2$ and $\Gamma; \Delta \vdash e : \tau_1$ for some τ_1 and τ_2. By type inversion rule 6, we have $\Gamma; \Delta \vdash v_1 : \tau_1$, which is what is required. The proof for rule A.52 is similar.

(Rule A.53) Here $e = \mathsf{case}_{\tau_1+\tau_2}\; v \;\mathsf{of}\; \mathsf{inl}(x : \tau_1) \Rightarrow e_1 \;\mathsf{orelse}\; \mathsf{inr}(x : \tau_2) \Rightarrow e_2 \;\mathsf{end}$, where $v = \mathsf{inl}_{\tau_1+\tau_2}(v')$. From type inversion rule 10, we learn that $\Gamma; \Delta \vdash v : \tau_1 + \tau_2$ and $\Gamma, x : \tau_1; \Delta \vdash e_1 : \tau$. Applying inversion again (rule 9) gives us $\Gamma; \Delta \vdash v' : \tau_1$. Using the substitution lemma (3), we have that $\Gamma; \Delta \vdash e_1[x := v'] : \tau$. The proof for rule A.54 is similar.

(Rule A.56) Here $e = \mathsf{if}\; v \;\mathsf{then}\; e_1 \;\mathsf{else}\; e_2 \;\mathsf{fi}$. From type inversion rule 11, we learn that $\Gamma; \Delta \vdash e_1 : \tau$. The proof for rule A.57 is similar.

(Rule A.55) Here $e = \mathsf{unroll}(\mathsf{roll}_\tau(v))$, $\Gamma; \Delta \vdash e : \tau'$, and $e' = v$. By type inversion rule 13, we have $\Delta \vdash \tau' \equiv p(\tau''[t := \mu t : \kappa.\tau''])(t) : \Omega$ and $\Gamma; \Delta \vdash \mathsf{roll}_\tau(v) : p(\mu t : \kappa.\tau'')(t)$; thus by type inversion rule refoldmu:inv:roll, we have $\Gamma; \Delta \vdash v : p(\tau''[t := \mu t : \kappa.\tau''])$.

(Rule A.58) Here $e = \mathrm{op}(e_1, e_2)$, with $e_1 \hookrightarrow e'_1$. By type inversion rule 3, we have $\Gamma; \Delta \vdash e : \mathsf{int}$, $\Gamma; \Delta \vdash e_1 : \mathsf{int}$, and $\Gamma; \Delta \vdash e_2 : \mathsf{int}$. By the induction hypothesis we have $\Gamma; \Delta \vdash e'_1 : \mathsf{int}$. Application of typing rule A.36, then, gives us $\Gamma; \Delta \vdash \mathrm{op}(e'_1, e_2) : \mathsf{int}$ as required.

(Rule A.70) Here $e = \mathsf{unroll}(e_1)$, $e' = \mathsf{unroll}(e'_1)$, $e_1 \hookrightarrow e'_1$, and $\Gamma; \Delta \vdash e : \tau$. By type inversion rule 13, we find that $\Gamma; \Delta \vdash e_1 : \mathsf{p}(\mu t : \kappa.\tau')(\mathsf{s})$ and $\Delta \vdash \tau \equiv \mathsf{p}(\tau'[t := \mu t : \kappa.\tau'])(\mathsf{s})$. By induction, $\Gamma; \Delta \vdash e'_1 : \mathsf{p}(\mu t : \kappa.\tau')(\mathsf{s})$, and therefore $\Gamma; \Delta \vdash \mathsf{unroll}(e'_1) : \mathsf{p}(\tau'[t := \mu t : \kappa.\tau'])(\mathsf{s})$ by typing rule A.47.

(Rule A.71) Here $e = \mathsf{roll}_\tau(e_1)$, $e' = \mathsf{roll}_\tau(e_1')$, $e_1 \hookrightarrow e_1'$, and $\Gamma; \Delta \vdash e : \tau'$. By type inversion rule 12, we find that $\Delta \vdash \tau \equiv \tau' \equiv (\mathsf{p}(\mu t : \kappa.\tau''))(\mathsf{s}) : \Omega$ and $\Gamma; \Delta \vdash e_1 : (\mathsf{p}(\tau''[t := \mu t : \kappa.\tau'']))(\mathsf{s})$. By induction, $\Gamma; \Delta \vdash e_1' : (\mathsf{p}(\tau''[t := \mu t : \kappa.\tau'']))(\mathsf{s})$, and hence $\Gamma; \Delta \vdash \mathsf{roll}_\tau(e_1') : (\mathsf{p}(\mu t : \kappa.\tau''))(\mathsf{s})$ by typing rule A.48.

(Other rules) The proofs of the remainder of the "search" rules (A.59 – A.68) are similar to the proof of rule A.58: use the type inversion lemma (4), apply the induction hypothesis, and re-apply a typing rule.

Proposition 6 (Canonical Forms). *Suppose that $v : \tau$ is a closed, well-formed value.*

1. *If $\vdash v : \mathsf{int}$, then $v = n$ for some n.*
2. *If $\vdash v : \mathsf{unit}$, then $v = ()$.*
3. *If $\vdash v : \tau_1 \to \tau_2$, then $v = \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}$ for some f, x, and e.*
4. *If $\vdash v : \tau_1 * \tau_2$, then $v = (v_1, v_2)$ for some v_1 and v_2.*
5. *If $\vdash v : \tau_1 + \tau_2$, then either $v = \mathsf{inl}_{\tau_1+\tau_2}(v_1)$ for some v_1 or $v = \mathsf{inr}_{\tau_1+\tau_2}(v_2)$ for some v_2.*
6. *Let $\mathsf{p} = \cdot \mid \pi_i$, and let $\mathsf{t} = \cdot \mid \tau'$. If $\vdash v : \mathsf{p}(\mu t : \kappa.\tau')(t)$, then $v = \mathsf{roll}_{\mathsf{p}(\mu t:\kappa.\tau)(t)}(v')$ for some v'.*

Theorem 7 (Progress). *If $\Gamma; \Delta \vdash e : \tau$, then either e is a value, or there exists e' such that $e \hookrightarrow e'$.*

Proof. The proof is by induction on the typing rules.

(Rule A.34) By application of the induction hypothesis.

(Rule A.35) n is a value.

(Rule A.36) $\Gamma; \Delta \vdash op(e_1, e_2) : \mathsf{int}$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values. Thus by evaluation rule A.49, $\mathrm{op}(e_1, e_2) \hookrightarrow e_1\ \mathrm{op}\ e_2$.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$, and using evaluation rule A.58 we have: $\mathrm{op}(e_1, e_2) \hookrightarrow \mathrm{op}(e_1', e_2)$.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$, and we can apply evaluation rule A.59.

(Rule A.37) $\mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}$ is a value.

(Rule A.38) $\Gamma; \Delta \vdash e_1(e_2) : \tau_2$, where $\Gamma; \Delta \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma; \Delta \vdash e_2 : \tau_1$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values. Using canonical forms, (Proposition 6), e_1 must be of the form $\mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e'\ \mathsf{end}$. Using A.50, $e \hookrightarrow e'[f := e_1, x := e_2]$.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$; use evaluation rule A.60.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$; use evaluation rule A.61.

(Rule A.39) The expression $()$ is already a value.

(Rule A.40) $\Gamma; \Delta \vdash (e_1, e_2) : \tau_1 * \tau_2$, where $\Gamma; \Delta \vdash e_1 : \tau_1$ and $\Gamma; \Delta \vdash e_2 : \tau_2$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values; thus (e_1, e_2) is itself a value.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$; use evaluation rule A.62.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$, and we can apply evaluation rule A.63.

(Rule A.42) $\Gamma; \Delta \vdash \mathsf{fst}(e) : \tau_1$, where $\Gamma; \Delta \vdash e : \tau_1 * \tau_2$. By the induction hypothesis, there are two cases:

- e is a value. Thus by canonical forms (Proposition 6), e is of the form (v_1, v_2). Use evaluation rule A.51.
- e is not a value. By the induction hypothesis, then, $e \hookrightarrow e'$; use evaluation rule A.64.

(Rule A.43) Same as rule A.42 above.

(Rule A.44) $\Gamma; \Delta \vdash \mathsf{inl}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2$, where $\Gamma; \Delta \vdash e : \tau_1$. By the induction hypothesis, there are two cases:

- e is a value; therefore $\mathsf{inl}_{\tau_1+\tau_2}(e)$ is itself a value.
- e is not a value. By the induction hypothesis, then, $e \hookrightarrow e'$; use evaluation rule A.66.

(Rule A.45) Same as rule A.44 above.

(Rule A.46) $\Gamma; \Delta \vdash \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x_1 : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inr}(x_2 : \tau_2) \Rightarrow e_2\ \mathsf{end} : \tau$, where $\Gamma; \Delta \vdash e : \tau_1 + \tau_2$, $\Gamma, x_1 : \tau_1; \Delta \vdash e_1 : \tau$, and $\Gamma, x_2 : \tau_2; \Delta \vdash e_2 : \tau$. By the induction hypothesis, there are two cases:

- e is a value. By canonical forms, either $e = \mathsf{inl}_{\tau_1+\tau_2}(v_1)$, or $e = \mathsf{inr}_{\tau_1+\tau_2}(v_2)$; use either evaluation rule A.53 or A.54, respectively.
- e is not a value. By the induction hypothesis, then, $e \hookrightarrow e'$; use evaluation rule A.68.

(Rule A.41) Here $e = \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$. There are two cases. If e is a value, then apply either rule A.56 or A.57. If e is not a value, then apply the search rule.

(Rule A.47) Here $e = \mathsf{unroll}(e_1)$, with $\Gamma; \Delta \vdash e_1 : \mathsf{p}(\mu t : \kappa.\tau)(\mathsf{s})$. By the induction hypothesis, there are two cases:

- e_1 is a value. Thus, by canonical forms, $e_1 = \mathsf{roll}_{\tau'}(v)$ for some v. Applying evaluation rule A.55 gives us $e \hookrightarrow v$.
- e_1 is not a value. Apply evaluation rule A.70.

(Rule A.48) Here $e = \mathsf{roll}_\tau(e_1)$, with $\Gamma; \Delta \vdash e_1 : \mathsf{p}(\tau'[t := \mu t : \kappa.\tau'])(\mathsf{s})$. By the induction hypothesis, there are two cases:

- e_1 is a value, therefore e is itself a value.
- e_1 is not a value. Apply evaluation rule A.71.

B Formal Properties of λ_{M2}

B.1 Static semantics

Environments We define two environments, Δ and Γ. The Δ environment is a *type environment* mapping type variables to their kinds, and Γ is a *term environment* mapping term variables to their types. Typing a term e with a type τ is denoted as $\Gamma; \Delta \vdash e : \tau$. The static semantics for these environments are as follows:

$$\vdash \Omega\ \mathit{kind} \qquad (B.1)$$

$$\frac{\vdash \kappa_1\ \mathit{kind} \quad \vdash \kappa_2\ \mathit{kind}}{\vdash \kappa_1 \Rightarrow \kappa_2\ \mathit{kind}} \qquad (B.2)$$

$$\frac{\vdash \kappa_1\ \mathit{kind} \quad \vdash \kappa_2\ \mathit{kind}}{\vdash \kappa_1 \circledast \kappa_2\ \mathit{kind}} \qquad (B.3)$$

$$\vdash \cdot\ \mathit{type\ env} \qquad (B.4)$$

$$\frac{\vdash \Delta\ \mathit{type\ env} \quad \vdash \kappa\ \mathit{kind}}{\vdash \Delta, t : \kappa\ \mathit{type\ env}} \qquad (B.5)$$

$$\Delta \vdash \cdot\ \mathit{term\ env} \qquad (B.6)$$

$$\frac{\Delta \vdash \Gamma\ \mathit{term\ env} \quad \Delta \vdash \tau : \Omega}{\Delta \vdash \Gamma, x : \tau\ \mathit{term\ env}} \qquad (B.7)$$

Type formation $\boxed{\Delta \vdash \tau : \kappa}$

$$\frac{t \in \Delta}{\Delta \vdash t : \Delta(t)} \qquad (B.8)$$

$$\frac{}{\Delta \vdash \mathsf{int} : \Omega} \qquad (B.9)$$

$$\frac{}{\Delta \vdash \mathsf{unit} : \Omega} \qquad (B.10)$$

$$\frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 \to \tau_2 : \Omega} \qquad (B.11)$$

$$\frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 + \tau_2 : \Omega} \qquad (B.12)$$

$$\frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 * \tau_2 : \Omega} \qquad (B.13)$$

$$\frac{\Delta, t : \kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda t{:}\kappa_1.\tau : \kappa_1 \Rightarrow \kappa_2} \qquad (B.14)$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \Rightarrow \kappa_2 \quad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2} \qquad (B.15)$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \quad \Delta \vdash \tau_2 : \kappa_2}{\Delta \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \circledast \kappa_2} \qquad (B.16)$$

$$\frac{\Delta \vdash \tau : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_1(\tau) : \kappa_1} \qquad (B.17)$$

$$\frac{\Delta \vdash \tau : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_2(\tau) : \kappa_2} \qquad (B.18)$$

$$\frac{\Delta \vdash f : \kappa \Rightarrow \kappa \quad \Delta \vdash g : \kappa \Rightarrow \Omega}{\Delta \vdash \mu_\kappa(f, g) : \Omega} \qquad (B.19)$$

$$\frac{\Delta \vdash \Gamma(x) : \Omega \quad (\forall x \in \mathrm{Dom}(\Gamma))}{\Delta \vdash \Gamma} \qquad (B.20)$$

Type equality $\boxed{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa}$

Rule B.21: reflexivity.

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau \equiv \tau : \kappa} \qquad (B.21)$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa}{\Delta \vdash \tau_2 \equiv \tau_1 : \kappa} \qquad (B.22)$$

Rule B.22: symmetry.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa \quad \Delta \vdash \tau_2 \equiv \tau_3 : \kappa}{\Delta \vdash \tau_1 \equiv \tau_3 : \kappa} \quad \text{(B.23)}$$

Rule B.23: transitivity.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Delta \vdash \tau_2 \equiv \tau_2' : \Omega}{\Delta \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \Omega} \quad \text{(B.24)}$$

We note here that the function type is only defined for types of kind Ω; see rule B.11.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Delta \vdash \tau_2 \equiv \tau_2' : \Omega}{\Delta \vdash \tau_1 + \tau_2 \equiv \tau_1' + \tau_2' : \Omega} \quad \text{(B.25)}$$

Note that the sum type is only defined for types of kind Ω; see rule B.12.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Delta \vdash \tau_2 \equiv \tau_2' : \Omega}{\Delta \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2' : \Omega} \quad \text{(B.26)}$$

Note, again, that the pair type is only defined for types of kind Ω; see rule B.13.

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \kappa_1 \quad \Delta \vdash \tau_2 \equiv \tau_2' : \kappa_2}{\Delta \vdash \langle \tau_1, \tau_2 \rangle \equiv \langle \tau_1', \tau_2' \rangle : \kappa_1 \circledast \kappa_2} \quad \text{(B.27)}$$

$$\frac{\Delta \vdash f \equiv f' : (\kappa \Rightarrow \kappa) \quad \Delta \vdash g \equiv g' : (\kappa \Rightarrow \Omega)}{\Delta \vdash \mu_\kappa(f, g) \equiv \mu_\kappa(f', g') : \Omega} \quad \text{(B.28)}$$

Note here the restriction on the kinds of f and g; this comes from type formation rule B.19.

$$\frac{\Delta \vdash \tau_1 \equiv \lambda t : \kappa.\tau : \kappa \Rightarrow \kappa' \quad \Delta \vdash \tau_2 : \kappa}{\Delta \vdash \tau_1 \tau_2 \equiv \tau_1[t := \tau_2] : \kappa'} \quad \text{(B.29)}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' : \kappa_1 \Rightarrow \kappa_2 \quad \Delta \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Delta \vdash \tau_1(\tau_2) \equiv \tau_1'(\tau_2') : \kappa_2} \quad \text{(B.30)}$$

$$\frac{\Delta, t : \kappa \vdash \tau \equiv \tau'[s := t] : \kappa' \quad t \mathrel{/\!\in} FV(\tau')}{\Delta \vdash \lambda t : \kappa.\tau \equiv \lambda s : \kappa.\tau' : \kappa \Rightarrow \kappa'} \quad \text{(B.31)}$$

$$\frac{\Delta \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_i(\langle \tau_1, \tau_2 \rangle) \equiv \tau_i : \kappa_i} \quad \text{(B.32)}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa_1 \circledast \kappa_2}{\Delta \vdash \pi_i(\tau_1) \equiv \pi_i(\tau_2) : \kappa_i} \quad \text{(B.33)}$$

Term formation

$$\boxed{\Gamma; \Delta \vdash e : \tau}$$

$$\frac{x \in \Gamma}{\Gamma; \Delta \vdash x : \Gamma(x)} \quad \text{(B.34)}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 \quad \Gamma; \Delta \vdash e : \tau_1}{\Gamma; \Delta \vdash e : \tau_2} \quad \text{(B.35)}$$

$$\frac{}{\Gamma; \Delta \vdash n : \mathsf{int}} \quad \text{(B.36)}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \mathsf{int} \quad \Gamma; \Delta \vdash e_2 : \mathsf{int}}{\Gamma; \Delta \vdash op(e_1, e_2) : \mathsf{int}} \quad \text{(B.37)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1; \ \Delta \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} : \ \tau_1 \to \tau_2} \quad \text{(B.38)}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta \vdash e_2 : \tau_1}{\Gamma; \Delta \vdash e_1(e_2) : \tau_2} \quad \text{(B.39)}$$

$$\frac{}{\Gamma; \Delta \vdash () : \mathsf{unit}} \quad \text{(B.40)}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \quad \Gamma; \Delta \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash (e_1, e_2) : \tau_1 * \tau_2} \quad \text{(B.41)}$$

$$\frac{\Gamma; \Delta \vdash e : \mathsf{int} \quad \Gamma; \Delta \vdash e_1 : \tau \quad \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} : \tau} \quad \text{(B.42)}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1 * \tau_2}{\Gamma; \Delta \vdash \mathsf{fst}(e) : \tau_1} \quad \text{(B.43)}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1 * \tau_2}{\Gamma; \Delta \vdash \mathsf{snd}(e) : \tau_2} \quad \text{(B.44)}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1}{\Gamma; \Delta \vdash \mathsf{inl}_{\tau_1 + \tau_2}(e) : \tau_1 + \tau_2} \quad \text{(B.45)}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathsf{inr}_{\tau_1 + \tau_2}(e) : \tau_1 + \tau_2} \quad \text{(B.46)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash e : \tau_1 + \tau_2 \\ \Gamma, x_1 : \tau_1; \Delta \vdash e_1 : \tau \\ \Gamma, x_2 : \tau_2; \Delta \vdash e_2 : \tau\end{array}}{\begin{array}{c}\Gamma; \Delta \vdash \mathsf{case}_{\tau_1 + \tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x_1 : \tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x_2 : \tau_2) \Rightarrow e_2\ \mathsf{end} : \tau\end{array}} \quad \text{(B.47)}$$

To do the roll/unroll in this language, we need a function which determines the type of the unrolled/rolled expression. This function is defined as:

$$\begin{array}{c}\mathcal{K}(\kappa, f : \kappa \Rightarrow \kappa, g : \kappa \Rightarrow \Omega) := \\ g(f(\mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, f))),\end{array}$$

where $\mathcal{H}$ is defined as follows:

$$\mathcal{H}(\kappa', \kappa, Q : \kappa \Rightarrow \kappa', f : \kappa \Rightarrow \kappa) :=$$
$$\begin{cases} \mu_\kappa(f, Q) & \text{when } \kappa = \Omega \\ \langle \mathcal{H}(\kappa_1, \kappa, \lambda h : \kappa.(\pi_1(Q\, h)), f), \\ \quad \mathcal{H}(\kappa_2, \kappa, \lambda h : \kappa.(\pi_2(Q\, h)), f)\rangle & \text{when } \kappa = \kappa_1 \circledast \kappa_2 \\ \lambda g : \kappa_1.\mathcal{H}(\kappa_2, \kappa, \lambda h : \kappa.(Q\, h)(g), f) & \text{when } \kappa = \kappa_1 \Rightarrow \kappa_2. \end{cases}$$

Now we can define roll and unroll:

$$\frac{\Delta \vdash \mu_\kappa(f,g) : \Omega \quad \Gamma; \Delta \vdash e : \mu_\kappa(f,g)}{\Gamma; \Delta \vdash \mathsf{unroll}(e) : \mathcal{K}(\kappa, f, g)} \tag{B.48}$$

$$\frac{\Delta \vdash \mu_\kappa(f,g) : \Omega \quad \Gamma; \Delta \vdash e : \mathcal{K}(\kappa, f, g)}{\Gamma; \Delta \vdash \mathsf{roll}_{\mu_\kappa(f,g)}(e) : \mu_\kappa(f,g)} \tag{B.49}$$

B.2 Dynamic semantics

The values of the language are as follows:

$$\begin{aligned} (\text{values})\ v ::= {} & n \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid \mathsf{inl}_{\tau_1+\tau_2}(v) \mid \mathsf{inr}_{\tau_1+\tau_2}(v) \\ & \mid () \mid (v_1, v_2) \mid \boldsymbol{\mathsf{roll}_{\mu_\kappa(f,g)}(v)} \end{aligned}$$

Primitive evaluation rules

$$\frac{}{\mathsf{op}(v_1, v_2) \hookrightarrow v_1\ \mathsf{op}\ v_2} \tag{B.50}$$

$$\frac{v_1 = \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}}{v_1(v_2) \hookrightarrow e[f := v_1, x := v_2]} \tag{B.51}$$

$$\frac{}{\mathsf{fst}(v_1, v_2) \hookrightarrow v_1} \tag{B.52}$$

$$\frac{}{\mathsf{snd}(v_1, v_2) \hookrightarrow v_2} \tag{B.53}$$

$$\frac{v = \mathsf{inl}_{\tau_1+\tau_2}(v')}{\begin{array}{c}\mathsf{case}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ \mathsf{inl}(x : \tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x : \tau_2) \Rightarrow e_2\ \mathsf{end} \\ \hookrightarrow e_1[x := v']\end{array}} \tag{B.54}$$

$$\frac{v = \mathsf{inr}_{\tau_1+\tau_2}(v')}{\begin{array}{c}\mathsf{case}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ \mathsf{inl}(x : \tau_1) \Rightarrow e_1 \\ \mathsf{orelse}\ \mathsf{inr}(x : \tau_2) \Rightarrow e_2\ \mathsf{end} \\ \hookrightarrow e_2[x := v']\end{array}} \tag{B.55}$$

$$\frac{v \mathrel{/}= 0}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_1} \tag{B.56}$$

$$\frac{v = 0}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_2} \tag{B.57}$$

$$\frac{}{\mathsf{unroll}(\mathsf{roll}_{\mu_\kappa(f,g)}(v)) \hookrightarrow v} \tag{B.58}$$

Search rules

$$\frac{e_1 \hookrightarrow e_1'}{\mathsf{op}(e_1, e_2) \hookrightarrow \mathsf{op}(e_1', e_2)} \tag{B.59}$$

$$\frac{e_2 \hookrightarrow e_2'}{\mathsf{op}(v, e_2) \hookrightarrow \mathsf{op}(v, e_2')} \tag{B.60}$$

$$\frac{e_1 \hookrightarrow e_1'}{e_1(e_2) \hookrightarrow e_1'(e_2)} \tag{B.61}$$

$$\frac{e_2 \hookrightarrow e_2'}{v(e_2) \hookrightarrow v(e_2')} \tag{B.62}$$

$$\frac{e_1 \hookrightarrow e_1'}{(e_1, e_2) \hookrightarrow (e_1', e_2)} \tag{B.63}$$

$$\frac{e_2 \hookrightarrow e_2'}{(v_1, e_2) \hookrightarrow (v_1, e_2')} \tag{B.64}$$

$$\frac{e \hookrightarrow e'}{\mathsf{fst}(e) \hookrightarrow \mathsf{fst}(e')} \tag{B.65}$$

$$\frac{e \hookrightarrow e'}{\mathsf{snd}(e) \hookrightarrow \mathsf{snd}(e')} \tag{B.66}$$

$$\frac{e \hookrightarrow e'}{\mathsf{inl}_{\tau_1+\tau_2}(e) \hookrightarrow \mathsf{inl}_{\tau_1+\tau_2}(e')} \tag{B.67}$$

$$\frac{e \hookrightarrow e'}{\mathsf{inr}_{\tau_1+\tau_2}(e) \hookrightarrow \mathsf{inr}_{\tau_1+\tau_2}(e')} \quad (B.68)$$

$$\frac{e \hookrightarrow e'}{\begin{array}{c}\mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x:\tau_1) \Rightarrow e_1\\ \mathsf{orelse}\ \mathsf{inr}(x:\tau_2) \Rightarrow e_2\ \mathsf{end}\\ \hookrightarrow \mathsf{case}_{\tau_1+\tau_2}\ e'\ \mathsf{of}\ \mathsf{inl}(x:\tau_1) \Rightarrow e_1\\ \mathsf{orelse}\ \mathsf{inr}(x:\tau_2) \Rightarrow e_2\ \mathsf{end}\end{array}} \quad (B.69)$$

$$\frac{e \hookrightarrow e'}{\begin{array}{c}\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}\\ \hookrightarrow \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}\end{array}} \quad (B.70)$$

$$\frac{e \hookrightarrow e'}{\mathsf{unroll}(e) \hookrightarrow \mathsf{unroll}(e')} \quad (B.71)$$

$$\frac{e \hookrightarrow e'}{\mathsf{roll}_{\mu_\kappa(f,g)}(e) \hookrightarrow \mathsf{roll}_{\mu_\kappa(f,g)}(e')} \quad (B.72)$$

B.3 Soundness and safety

Properties of static semantics

Proposition 8 (Strong normalization). *Type reduction for the language given above is strongly normalizing.*

Proof (sketch). Rules B.21 through B.33 describe a simply-typed lambda calculus with pair types and constants. Proofs of strong normalization for $\lambda \rightarrow$ are familiar in the literature (see [18], for instance.)

Corollary 9 (Decidability). *Type-checking the language given above is decidable.*

Proof. Judgments for the formations of kinds (B.1–B.3), type environments (B.4–B.5), term environments (B.6–B.7), and type formation are all syntax-directed and hence decidable.

Type equivalence, however, is not syntax-directed. Since type reductions are strongly normalizing (Proposition 8), we can determine whether $\tau_1 \equiv \tau_2$ by reducing both τ_1 and τ_2 to normal form, then testing whether the normal forms are syntactically congruent.

Term formation is syntax-directed, excepting the type-equivalence rule B.35. However, if the type-checker always reduces types to normal form, then rule B.35 can be omitted.

Soundness

Lemma 10 (Substitution).

1. *If* $\Gamma; \Delta \vdash e' : \tau'$ *and* $\Gamma, x : \tau'; \Delta \vdash e : \tau$, *then* $\Gamma; \Delta \vdash e[x := e'] : \tau$.
2. *If* $\Delta \vdash \tau' : \kappa$ *and* $\Gamma; \Delta, t : \kappa \vdash e : \tau$, *then* $\Gamma[t := \tau']; \Delta \vdash e[t := \tau'] : \tau[t := \tau']$.

Proof.

1. By induction on the derivation of $\Gamma, x : \tau'; \Delta \vdash e : \tau$.
2. By induction on the derivation of $\Gamma; \Delta, t : \kappa \vdash e : \tau$.

Lemma 11 (Type Inversion).

1. *If* $\Gamma; \Delta \vdash x : \tau$, *then* $\Delta \vdash \Gamma(x) \equiv \tau : \Omega$.
2. *If* $\Gamma; \Delta \vdash n : \tau$, *then* $\tau \equiv \mathsf{int} : \Omega$.
3. *If* $\Gamma; \Delta \vdash op(e_1, e_2) : \tau$, *then* $\Delta \vdash \tau \equiv \mathsf{int} : \Omega$, $\Gamma; \Delta \vdash e_1 : \mathsf{int}$, *and* $\Gamma; \Delta \vdash e_2 : \mathsf{int}$.

4. *If* $\Gamma;\Delta \vdash \textsf{fun}\ f(x:\tau_1):\tau_2\ \textsf{is}\ e\ \textsf{end} : \tau$, *then* $\Gamma, f:\tau_1 \to \tau_2, x:\tau_1;\Delta \vdash e:\tau_2$ *and* $\Delta \vdash \tau \equiv \tau_1 \to \tau_2 : \Omega$.
5. *If* $\Gamma;\Delta \vdash e_1(e_2):\tau$, *then there exists* τ_2 *such that* $\Gamma;\Delta \vdash e_1 : \tau_2 \to \tau$ *and* $\Gamma;\Delta \vdash e_2 : \tau_2$.
6. *If* $\Gamma;\Delta \vdash (e_1, e_2) : \tau$, *then there exist* τ_1 *and* τ_2 *such that* $\Delta \vdash \tau \equiv \tau_1 * \tau_2 : \Omega$, $\Gamma;\Delta \vdash e_1 : \tau_1$, *and* $\Gamma;\Delta \vdash e_2 : \tau_2$.
7. *If* $\Gamma;\Delta \vdash () : \tau$, *then* $\Delta \vdash \tau \equiv \textsf{unit} : \Omega$.
8. *If* $\Gamma;\Delta \vdash \textsf{fst}(e) : \tau$, *then there exists* τ_2 *such that* $\Gamma;\Delta \vdash e : \tau * \tau_2$, *and similarly for* $\textsf{snd}$.
9. *If* $\Gamma;\Delta \vdash \textsf{inl}_{\tau_1+\tau_2}(e_1) : \tau$, *then* $\Delta \vdash \tau \equiv \tau_1 + \tau_2 : \Omega$ *and* $\Gamma;\Delta \vdash e_1 : \tau_1$. *The* $\textsf{inr}$ *case is similar.*
10. *If* $\Gamma;\Delta \vdash \textsf{case}_{\tau_1+\tau_2}\ e\ \textsf{of inl}(x:\tau_1) \Rightarrow e_1\ \textsf{orelse inr}(x:\tau_2) \Rightarrow e_2\ \textsf{end} : \tau$, *then* $\Gamma;\Delta \vdash e : \tau_1 + \tau_2$, $\Gamma, x_1 : \tau_1;\Delta \vdash e_1 : \tau$, *and* $\Gamma, x_2 : \tau_2;\Delta \vdash e_2 : \tau$.
11. *If* $\Gamma;\Delta \vdash \textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2\ \textsf{fi} : \tau$, *then* $\Gamma;\Delta \vdash e : \textsf{int}$, $\Gamma;\Delta \vdash e_1 : \tau$, *and* $\Gamma;\Delta \vdash e_2 : \tau$.
12. *If* $\Gamma;\Delta \vdash \textsf{roll}_{\mu_\kappa(f,g)}(e) : \tau$, *then* $\Delta \vdash \tau \equiv \mu_\kappa(f,g) : \Omega$ *and* $\Gamma;\Delta \vdash e : \mathcal{K}(\kappa, f, g)$.
13. *If* $\Gamma;\Delta \vdash \textsf{unroll}(e) : \tau$, *then there exist* f, g, *and* κ *such that* $\Delta \vdash \tau \equiv \mathcal{K}(\kappa, f, g) : \Omega$, *and* $\Gamma;\Delta \vdash e : \mu_\kappa(f,g)$.

Proof (sketch). The proof relies on the fact that every typing derivation $\Gamma;\Delta \vdash e:\tau$ is the result of a unique typing rule, apart from the type-equality rule B.35. Since the type-equality rule has only one premise, each typing derivation ending with $\Gamma;\Delta \vdash e:\tau$ is a finite chain of zero or more type-equality judgments, with a necessarily unique type judgment at the top.

Theorem 12 (Preservation). *If* $\Gamma;\Delta \vdash e:\tau$ *and* $e \hookrightarrow e'$, *then* $\Gamma;\Delta \vdash e' : \tau$.

Proof. The proof is by induction on the rules defining one-step evaluation.

(Rule B.50) Here $e = \text{op}(v_1, v_2)$ and $e' = v_1\ \text{op}\ v_2$. By type inversion rule 3, $\Delta \vdash \tau \equiv \textsf{int} : \Omega$, $\Gamma;\Delta \vdash v_1 : \textsf{int}$, and $\Gamma;\Delta \vdash v_2 : \textsf{int}$. The meta-level integer operation "op" returns an integer, so $\Gamma;\Delta \vdash e' : \textsf{int}$.

(Rule B.51) Here $e = v_1(v_2)$, where $v_1 = \textsf{fun}\ f(x:\tau_1):\tau_2\ \textsf{is}\ e_1\ \textsf{end}$. From syntax and type inversion rule 5, we know that $\Gamma;\Delta \vdash e:\tau_2$, and that $\Gamma;\Delta \vdash v_1 : \tau_1 \to \tau_2$. Using inversion again (rule 4), we have that $\Gamma, f:\tau_1 \to \tau_2, x:\tau_1;\Delta \vdash e_1 : \tau_2$. By twice applying the substitution lemma (10), we have that $\Gamma;\Delta \vdash e_1[f := v_1, x := v_2] : \tau_2$, which is what is required.

(Rule B.52) Here $e = \textsf{fst}(v_1, v_2)$. By type inversion rule 8, we have $\Gamma;\Delta \vdash (v_1, v_2) : \tau_1 * \tau_2$ and $\Gamma;\Delta \vdash e : \tau_1$ for some τ_1 and τ_2. By type inversion rule 6, we have $\Gamma;\Delta \vdash v_1 : \tau_1$, which is what is required. The proof for rule B.53 is similar.

(Rule B.54) Here $e = \textsf{case}_{\tau_1+\tau_2}\ v\ \textsf{of inl}(x:\tau_1) \Rightarrow e_1\ \textsf{orelse inr}(x:\tau_2) \Rightarrow e_2\ \textsf{end}$, where $v = \textsf{inl}_{\tau_1+\tau_2}(v')$. From type inversion rule 10, we learn that $\Gamma;\Delta \vdash v : \tau_1 + \tau_2$ and $\Gamma, x:\tau_1;\Delta \vdash e_1 : \tau$. Applying inversion again (rule 9) gives us $\Gamma;\Delta \vdash v' : \tau_1$. Using the substitution lemma (10), we have that $\Gamma;\Delta \vdash e_1[x := v'] : \tau$. The proof for rule B.55 is similar.

(Rule B.58) Here $e = \mathsf{unroll}(\mathsf{roll}_{\mu_\kappa(f,g)}(v))$ and $e' = v$. By type inversion rule 13, we have that $\mathsf{roll}_{\mu_\kappa(f,g)}(v) : \mu_\kappa(f,g)$ and that $\Delta \vdash \tau \equiv \mathcal{K}(\kappa, f, g) : \Omega$. By type inversion rule 12, we have that $v : \mathcal{K}(\kappa, f, g)$, and therefore from type equality we have $v : \tau$.

(Rule B.59) Here $e = \mathrm{op}(e_1, e_2)$, with $e_1 \hookrightarrow e_1'$. By type inversion rule 3, we have $\Gamma; \Delta \vdash e : \mathsf{int}$, $\Gamma; \Delta \vdash e_1 : \mathsf{int}$, and $\Gamma; \Delta \vdash e_2 : \mathsf{int}$. By the induction hypothesis we have $\Gamma; \Delta \vdash e_1' : \mathsf{int}$. Application of typing rule B.37, then, gives us $\Gamma; \Delta \vdash \mathrm{op}(e_1', e_2) : \mathsf{int}$ as required.

(Rule B.56) Here $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$. From type inversion rule 11, we learn that $\Gamma; \Delta \vdash e_1 : \tau$. The proof for rule B.57 is similar.

(Rule B.71) Here $e = \mathsf{unroll}(e_1)$, $e' = \mathsf{unroll}(e_1')$, $e_1 \hookrightarrow e_1'$, and $\Gamma; \Delta \vdash \mathsf{unroll}(e_1) : \tau$. By type inversion rule 13, we find that $\Gamma; \Delta \vdash e_1 : \mu_\kappa(f,g)$, $\Delta \vdash \tau \equiv \mathcal{K}(\kappa, f, g) : \Omega$, and hence by induction that $\Gamma; \Delta \vdash e_1' : \mu_\kappa(f,g)$. Thus by typing rule B.48, $\Gamma; \Delta \vdash \mathsf{unroll}(e_1') : \mathcal{K}(\kappa, f, g) \equiv \tau$.

(Rule B.72) Here $e = \mathsf{roll}_{\mu_\kappa(f,g)}(e_1)$, $e' = \mathsf{roll}_{\mu_\kappa(f,g)}(e_1')$, $e_1 \hookrightarrow e_1'$, and $\Gamma; \Delta \vdash \mathsf{roll}_{\mu_\kappa(f,g)}(e_1) : \tau$. By type inversion rule 12, we find that $\Gamma; \Delta \vdash e_1 : \mathcal{K}(\kappa, f, g)$, and hence by induction that $\Gamma; \Delta \vdash e_1' : \mathcal{K}(\kappa, f, g)$. By the application of typing rule B.49, we have $\Gamma; \Delta \vdash \mathsf{roll}_{\mu_\kappa(f,g)}(e_1') : \mu_\kappa(f,g) \equiv \tau$.

(Other rules) The proofs of the remainder of the "search" rules (B.60 – B.69) are similar to the proof of rule B.59: use the type inversion lemma (11), apply the induction hypothesis, and re-apply a typing rule.

Proposition 13 (Canonical Forms). *Suppose that $v : \tau$ is a closed, well-formed value.*

1. *If $\vdash v : \mathsf{int}$, then $v = n$ for some n.*
2. *If $\vdash v : \mathsf{unit}$, then $v = ()$.*
3. *If $\vdash v : \tau_1 \to \tau_2$, then $v = \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}$ for some f, x, and e.*
4. *If $\vdash v : \tau_1 * \tau_2$, then $v = (v_1, v_2)$ for some v_1 and v_2.*
5. *If $\vdash v : \tau_1 + \tau_2$, then either $v = \mathsf{inl}_{\tau_1+\tau_2}(v_1)$ for some v_1 or $v = \mathsf{inr}_{\tau_1+\tau_2}(v_2)$ for some v_2.*
6. *If $\vdash v : \mu_\kappa(f,g)$, then $v = \mathsf{roll}_{\mu_\kappa(f,g)}(v')$ for some v'.*

Theorem 14 (Progress). *If $\Gamma; \Delta \vdash e : \tau$, then either e is a value, or there exists e' such that $e \hookrightarrow e'$.*

Proof. The proof is by induction on the typing rules.

(Rule B.35) By application of the induction hypothesis.

(Rule B.36) n is a value.

(Rule B.37) $\Gamma; \Delta \vdash op(e_1, e_2) : \mathsf{int}$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values. Thus by evaluation rule B.50, $\mathrm{op}(e_1, e_2) \hookrightarrow e_1$ op e_2.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$, and using evaluation rule B.59 we have: $\mathrm{op}(e_1, e_2) \hookrightarrow \mathrm{op}(e_1', e_2)$.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$, and we can apply evaluation rule B.60.

(Rule B.38) $\mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}$ is a value.

(Rule B.39) $\Gamma; \Delta \vdash e_1(e_2) : \tau_2$, where $\Gamma; \Delta \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma; \Delta \vdash e_2 : \tau_1$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values. Using canonical forms, (Proposition 13), e_1 must be of the form $\mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e'\ \mathsf{end}$. Using B.51, $e \hookrightarrow e'[f := e_1, x := e_2]$.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$; use evaluation rule B.61.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$; use evaluation rule B.62.

(Rule B.40) The expression () is already a value.

(Rule B.41) $\Gamma; \Delta \vdash (e_1, e_2) : \tau_1 * \tau_2$, where $\Gamma; \Delta \vdash e_1 : \tau_1$ and $\Gamma; \Delta \vdash e_2 : \tau_2$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values; thus (e_1, e_2) is itself a value.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$; use evaluation rule B.63.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$, and we can apply evaluation rule B.64.

(Rule B.43) $\Gamma; \Delta \vdash \mathsf{fst}(e) : \tau_1$, where $\Gamma; \Delta \vdash e : \tau_1 * \tau_2$. By the induction hypothesis, there are two cases:

- e is a value. Thus by canonical forms (Proposition 13), e is of the form (v_1, v_2). Use evaluation rule B.52.
- e is not a value. By the induction hypothesis, then, $e \hookrightarrow e'$; use evaluation rule B.65.

(Rule B.44) Same as rule B.43 above.

(Rule B.45) $\Gamma; \Delta \vdash \mathsf{inl}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2$, where $\Gamma; \Delta \vdash e : \tau_1$. By the induction hypothesis, there are two cases:

- e is a value; therefore $\mathsf{inl}_{\tau_1+\tau_2}(e)$ is itself a value.
- e is not a value. By the induction hypothesis, then, $e \hookrightarrow e'$; use evaluation rule B.67.

(Rule B.46) Same as rule B.45 above.

(Rule B.47) $\Gamma;\Delta \vdash \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x_1 : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inr}(x_2 : \tau_2) \Rightarrow e_2\ \mathsf{end} : \tau$, where $\Gamma;\Delta \vdash e : \tau_1 + \tau_2$, $\Gamma, x_1 : \tau_1;\Delta \vdash e_1 : \tau$, and $\Gamma, x_2 : \tau_2;\Delta \vdash e_2 : \tau$. By the induction hypothesis, there are two cases:

- e is a value. By canonical forms, either $e = \mathsf{inl}_{\tau_1+\tau_2}(v_1)$, or $e = \mathsf{inr}_{\tau_1+\tau_2}(v_2)$; use either evaluation rule B.54 or B.55, respectively.
- e is not a value. By the induction hypothesis, then, $e \hookrightarrow e'$; use evaluation rule B.69.

(Rule B.42) Here $e = \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$. There are two cases. If e is a value, then apply either rule B.56 or B.57. If e is not a value, then apply the search rule.

(Rule B.48) Here $e = \mathsf{unroll}(e_1)$, with $e_1 : \mu_\kappa(f, g)$. By the induction hypothesis, there are two cases:

- e_1 is a value. Thus, by canonical forms, $e_1 = \mathsf{roll}_{\mu_\kappa(f,g)}(v)$ for some v. Applying evaluation rule B.58 gives us $e \hookrightarrow v$.
- e_1 is not a value. Apply evaluation rule B.71.

(Rule B.49) Here $e = \mathsf{roll}_{\mu_\kappa(f,g)}(e_1)$ with $e_1 : \mathcal{K}(\kappa, f, g)$. By the induction hypothesis, there are two cases:

- e_1 is a value, therefore $e = \mathsf{roll}_{\mu_\kappa(f,g)}(e_1)$ is itself a value.
- e_1 is not a value. Apply evaluation rule B.72.

C Properties of Translation from λ_{M1} to λ_{M2}

C.1 Well-typedness of the translation

Lemma 15. *Let* $\mathsf{p} = \cdot \mid \pi_i$. *Let* $\tau = \mathcal{H}(\kappa', \kappa, Q, f)$. *If* $\Delta \vdash (p(\tau))(\tau') : \Omega$, *then* $\Delta \vdash (p(\tau))(\tau') \equiv \mu_\kappa(f, g) : \Omega$, *with* $\Delta \vdash g \equiv \lambda k : \kappa.(p(Q\,k))([\![\tau']\!]) : \kappa \Rightarrow \Omega$.

Proof. The proof is by cases on p. If $\mathsf{p} = \cdot$, then $\kappa' = \kappa'' \Rightarrow \Omega$, and $\mathcal{H}(\kappa', \kappa, Q, f) = \lambda g : \kappa''.\mathcal{H}(\Omega, \kappa, \lambda h : \kappa.(Q\,h)(g), f)$, which reduces to

$$\lambda g : \kappa''.\mu_\kappa(f, \lambda h : \kappa.(Q\,h)(g)).$$

When applied to $[\![\tau']\!]$ this is equivalent to (using type equality rule B.29)

$$\mu_\kappa(f, \lambda h : \kappa.(Q\,h)([\![\tau']\!]),$$

which is what is required.

If $\mathsf{p} = \pi_i$, then typing dictates that $\kappa' = \kappa_1 \circledast \kappa_2$. Applying $\mathcal{H}$ gives:

$$\tau = \langle \mathcal{H}(\kappa_1, \kappa, \lambda h : \kappa.(\pi_1(Q\,h)), f),\ \mathcal{H}(\kappa_2, \kappa, \lambda h : \kappa.(\pi_2(Q\,h)), f) \rangle\,.$$

We have that $(\mathsf{p}(\tau))([\![\tau']\!])$ is equivalent to (using type equality rule B.32):

$$(\mathcal{H}(\kappa_i, \kappa, \lambda h : \kappa.(\pi_i(Q\,h)), f)))([\![\tau']\!]).$$

By induction, this expression is equivalent to $\mu_\kappa(f, g)$, with $g = \lambda k : \kappa.((\lambda h : \kappa.\pi_i(Q\,h))(k)))([\![\tau']\!])$. This β-reduces to $\lambda k : \kappa.(\pi_i(Q\,k)))([\![\tau']\!])$, which is what is required.

Corollary 16. *Let $p = \cdot \mid \pi_i$ and $\tau = \mathcal{H}(\kappa', \kappa, Q, f)$. If $\Delta \vdash p(\tau) : \Omega$, then $\Delta \vdash p(\tau) \equiv \mu_\kappa(f, g) : \Omega$, with $\Delta \vdash g \equiv \lambda k : \kappa.p(Q\,k) : \kappa \Rightarrow \Omega$.*

Proof. Similar to lemma 15.

Theorem 17. *If $\Gamma; \Delta \vdash e : \tau$, then $[\![\Gamma]\!]\,; \Delta \vdash [\![e]\!] : [\![\tau]\!]$.*

Proof. We prove by induction on the typing derivations. The only interesting cases are for rules A.47 and A.48.

(Case A.47) Here $\Gamma; \Delta \vdash \mathsf{unroll}(e) : \mathsf{p}(\tau'[t := \mu t : \kappa.\tau'])(\mathsf{s})$, with premises $\Gamma; \Delta \vdash e : \mathsf{p}(\mu t : \kappa.\tau')(\mathsf{s})$ and $\Delta \vdash \mathsf{p}(\mu t : \kappa.\tau')(\mathsf{s}) : \Omega$. By induction we have

$$[\![\Gamma]\!]\,; \Delta \vdash [\![e]\!] : \mathsf{p}(\mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!]))([\![\mathsf{s}]\!]).$$

From Lemma 15 or Corollary 16 (whichever is applicable depending on the value of s), we have that

$$\Delta \vdash \mathsf{p}(\mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!]))([\![\mathsf{s}]\!]) \equiv \mu_\kappa(\lambda t : \kappa.\,[\![\tau']\!]\,, \lambda k : \kappa.\mathsf{p}(k)([\![\mathsf{s}]\!])) : \Omega.$$

The typing rule for unroll in the destination language is:

$$\frac{\Delta \vdash \mu_\kappa(f, g) : \Omega \quad \Gamma; \Delta \vdash e : \mu_\kappa(f, g)}{\Gamma; \Delta \vdash \mathsf{unroll}(e) : \mathcal{K}(\kappa, f, g)}.$$

In order to type the translated expression, we need to evaluate $K(\kappa, f, g)$. Filling in our values for f and g, we get

$$K(\kappa, \lambda t : \kappa.\,[\![\tau']\!]\,, \lambda k : \kappa.\mathsf{p}(k)([\![\mathsf{s}]\!])) = \big(\lambda k : \kappa.\mathsf{p}(k)([\![\mathsf{s}]\!])\big)\big((\lambda t : \kappa.\,[\![\tau']\!])(\mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!]))\big),$$

which reduces to

$$\mathsf{p}([\![\tau']\!]\,[t := \mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!])])([\![\mathsf{s}]\!]).$$

If we translate the type $\tau = \mathsf{p}(\tau'[t := \mu t : \kappa.\tau'])(\mathsf{s})$, we obtain $\mathsf{p}([\![\tau']\!]\,[t := \mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!])])([\![\mathsf{s}]\!])$, which matches our value for $\mathcal{K}(\kappa, f, g)$. Thus by typing rule B.48, we have that $[\![\Gamma]\!]\,; \Delta \vdash \mathsf{unroll}([\![e]\!]) : [\![\tau]\!]$.

(Case A.48) Here $\Gamma; \Delta \vdash \mathsf{roll}_\tau(e)$, where $\Delta \vdash \tau \equiv \mathsf{p}(\mu t : \kappa.\tau)(\mathsf{s}) : \Omega$ and $\Gamma; \Delta \vdash e : \mathsf{p}(\tau'[t := \mu t : \kappa.\tau'])(\mathsf{s})$. By induction, we have that

$$[\![\Gamma]\!]\,; \Delta \vdash [\![e]\!] : \mathsf{p}([\![\tau']\!]\,[t := \mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!])])([\![\mathsf{s}]\!]).$$

From corollary 16, we have that

$$\Delta \vdash \mathsf{p}(\mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!]))([\![\mathsf{s}]\!]) \equiv \mu_\kappa(\lambda t : \kappa.\,[\![\tau']\!]\,, \lambda k : \kappa.\mathsf{p}(k)([\![\mathsf{s}]\!])) : \Omega.$$

Hence, we can compute $\mathcal{K}(\kappa, f, g)$ as in the proof for case A.47, yielding:

$$\Delta \vdash \mathcal{K}(\kappa, \lambda t : \kappa.\,[\![\tau']\!]\,, \lambda k : \kappa.\mathsf{p}(k)([\![\mathsf{s}]\!])) \equiv \mathsf{p}([\![\tau']\!]\,[t := \mathcal{H}(\kappa, \kappa, \lambda x : \kappa.x, \lambda t : \kappa.\,[\![\tau']\!])])([\![\mathsf{s}]\!]) : \Omega.$$

Thus, $[\![\Gamma]\!]\,; \Delta \vdash [\![e]\!] : \mathcal{K}(\kappa, f, g)$, and by typing rule B.49 we have $[\![\Gamma]\!]\,; \Delta \vdash \mathsf{roll}_{\mu_\kappa(f,g)}([\![e]\!]) : [\![\mathsf{p}(\mu t : \kappa.\tau')(\mathsf{s})]\!]$.

D Formal properties of λ_i^R

D.1 Dynamic Semantics

We present a small-step call-by-value operational semantics for λ_i^R. The values are defined as follows:

$$\begin{array}{lrl} \text{(value)} & v ::= & n \mid (v_1, v_2) \mid \mathsf{roll}_\tau(e) \mid \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(v) \\ & & \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \end{array}$$

The reduction relation $\hookrightarrow$ is specified by the following primitive evaluation rules:

$$\frac{}{op(v_1, v_2) \hookrightarrow v_1\ op\ v_2} \tag{D.1}$$

$$\frac{v_1 = \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}}{v_1(v_2) \hookrightarrow e[f := v_1, x := v_2]} \tag{D.2}$$

$$\frac{v\ /= 0}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_1} \tag{D.3}$$

$$\frac{v = 0}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_2} \tag{D.4}$$

$$\frac{}{\#i(v_1, v_2) \hookrightarrow v_i} \tag{D.5}$$

$$\frac{v = \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(v')}{\mathsf{case}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{or}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2 \hookrightarrow e_i[x := v']} \tag{D.6}$$

$$\frac{}{\mathsf{unroll}(\mathsf{roll}_\tau(v)) \hookrightarrow v} \tag{D.7}$$

The "search rules" have been omitted for brevity.

D.2 Static Semantics

Environments We define an environment Γ, which is a *term environment* mapping term variables to their types. Typing a term e with a type τ is denoted as $\Gamma \vdash e : \tau$. The typechecking is assumed to occur within a CIC environment Δ. The static semantics for this environment are as follows:

$$\Delta \vdash \cdot\ \textit{term env} \tag{D.8}$$

$$\frac{\Delta \vdash \Gamma\ \textit{term env} \quad \Delta \vdash \tau : \Omega}{\Delta \vdash \Gamma, x : \tau\ \textit{term env}} \tag{D.9}$$

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

$$\frac{x \in \Gamma}{\Gamma;\Delta \vdash x : \Gamma(x)} \tag{D.10}$$

The following rule deals with type equality. We take equality on types from the CIC $\beta\eta\iota$-equality on CIC terms.

$$\frac{\vdash \tau_1 \equiv \tau_2 \quad \Gamma;\Delta \vdash e : \tau_1}{\Gamma;\Delta \vdash e : \tau_2} \tag{D.11}$$

$$\frac{}{\Gamma;\Delta \vdash n : \mathsf{int}} \tag{D.12}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \mathsf{int} \quad \Gamma;\Delta \vdash e_2 : \mathsf{int}}{\Gamma;\Delta \vdash op(e_1, e_2) : \mathsf{int}} \tag{D.13}$$

$$\frac{\Gamma;\Delta \vdash e : \mathsf{int} \quad \Gamma;\Delta \vdash e_1 : \tau \quad \Gamma;\Delta \vdash e_2 : \tau}{\Gamma;\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \tag{D.14}$$

$$\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1;\Delta \vdash e : \tau_2}{\Gamma;\Delta \vdash \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} :\ \tau_1 \rightarrow \tau_2} \tag{D.15}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma;\Delta \vdash e_2 : \tau_1}{\Gamma;\Delta \vdash e_1(e_2) : \tau_2} \tag{D.16}$$

$$\frac{}{\Gamma;\Delta \vdash () : \mathsf{unit}} \tag{D.17}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \tau_1 \quad \Gamma;\Delta \vdash e_2 : \tau_2}{\Gamma;\Delta \vdash (e_1, e_2) : \tau_1 * \tau_2} \tag{D.18}$$

$$\frac{\Gamma;\Delta \vdash e : \tau_1 * \tau_2 \quad i \in \{1, 2\}}{\Gamma;\Delta \vdash \#i(e) : \tau_i} \tag{D.19}$$

$$\frac{\Gamma;\Delta \vdash e : \tau_i \quad i \in \{1, 2\}}{\Gamma;\Delta \vdash \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2} \tag{D.20}$$

$$\frac{\Gamma;\Delta \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1;\Delta \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2;\Delta \vdash e_2 : \tau}{\Gamma;\Delta \vdash \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inj}^{(1)}(x_1 : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inj}^{(2)}(x_2 : \tau_2) \Rightarrow e_2\ \mathsf{end} : \tau} \tag{D.21}$$

To do the roll/unroll in this language, we need a function which determines the type of the unrolled/rolled expression. This function is defined as:

$$\begin{aligned}&K[\kappa : \mathtt{Kind}; f : \kappa \Rightarrow \kappa; g : \kappa \Rightarrow \Omega] := \\ &\quad g(f(H(\kappa, \kappa, \lambda x : \kappa.x, f))).\end{aligned}$$

where H is defined as follows:

$$
\begin{array}{l}
H[\kappa' : \texttt{Kind}; \kappa : \texttt{Kind}; Q : \kappa \Rightarrow \kappa'; f : \kappa \Rightarrow \kappa] := \\
\quad \textsf{Cases } \kappa' \text{ of} \\
\qquad \Omega \longrightarrow \mu_\kappa(f, Q) \\
\quad \mid \ \kappa_1 \circledast \kappa_2 \longrightarrow \langle H(\kappa_1, \kappa, \lambda h : \kappa.(\textsf{fst}(Q\,h)), f),\ H(\kappa_2, \kappa, \lambda h : \kappa.(\textsf{snd}(Q\,h)), f)\rangle \\
\quad \mid \ \kappa_1 \Rightarrow \kappa_2 \longrightarrow \lambda g : \kappa_1.H(\kappa_2, \kappa, \lambda h : \kappa.(Q\,h)(g), f)
\end{array}
$$

To clarify the presentation, the formulation of these functions is not in CIC syntax; a few details have been elided for sake of presentation. Firstly, the variables κ and κ' above are actually dependently typed on `toSet` k, where k belongs to `Kind` mentioned earlier. This is what allows us to case over the kind constructors. Secondly, in order for the COQ typechecker to unify κ' with the branches of the Cases statement, we need to use a dependent case construct and abstract over Q. The details of this, however, are somewhat technical and do not help to illuminate the key issue. COQ code for these examples is in Figure 5.

Given this type-unrolling function, we can now define roll and unroll:

$$\frac{\Gamma; \Delta \vdash e : \mu_{\texttt{toSet}(\kappa)}(f, g)}{\Gamma; \Delta \vdash \textsf{unroll}(e) : (K\,\kappa\,f\,g)} \tag{D.22}$$

$$\frac{\Gamma; \Delta \vdash e : (K\,\kappa\,f\,g)}{\Gamma; \Delta \vdash \textsf{roll}_{\mu_{\texttt{toSet}(\kappa)}(f,g)}(e) : \mu_{\texttt{toSet}(\kappa)}(f, g)} \tag{D.23}$$

D.3 Properties of static semantics

Proposition 18 (Strong normalization). *Type reduction for the language given above is strongly normalizing.*

Proof. We are using CIC *so the proof is from Werner [19].*

Corollary 19 (Decidability). *Type-checking the language given above is decidable.*

Proof. Judgments for the formations of term environments (D.8–D.9) and type formation are all syntax-directed and hence decidable.

Type equivalence, however, is not syntax-directed. Since type reductions are strongly normalizing (Proposition 18), we can determine whether $\tau_1 \equiv \tau_2$ by reducing both τ_1 and τ_2 to normal form, then testing whether the normal forms are syntactically congruent.

Term formation is syntax-directed, excepting the type-equivalence rule D.11. However, if the type-checker always reduces types to normal form, then rule D.11 can be omitted. □

```
Inductive Omega: Set :=
   Int: Omega
 | Unit: Omega
 | To: Omega -> Omega -> Omega
 | PairType: Omega -> Omega -> Omega
 | SumType: Omega -> Omega -> Omega
 | Mu: (K:Set) (K->K)->(K->Omega)->Omega.

Inductive Kind: Set :=
   omega: Kind
 | TO: Kind -> Kind -> Kind
 | PairKind: Kind -> Kind -> Kind.


Fixpoint toSet [K:Kind] : Set :=
    Cases K of
        omega            => Omega
      | (TO k1 k2)       => (toSet k1)->(toSet k2)
      | (PairKind k1 k2) => (toSet k1) * (toSet k2)
    end.


Fixpoint H [k':Kind]: (k:Kind)(((toSet k) -> (toSet k'))
                     -> ((toSet k) -> (toSet k)) -> (toSet k')) :=
  [k:Kind][Q:((toSet k) -> (toSet k')); f:((toSet k) -> (toSet k))]
  ((<[k':Kind](((toSet k)->(toSet k'))->(toSet k'))>Cases k' of
    omega            => [QQ:((toSet k) -> Omega)](Mu (toSet k) f QQ)
  | (PairKind k1 k2)=> [QQ:((toSet k) -> ((toSet k1) * (toSet k2)))]
                        ((H k1 k ([h:(toSet k)] (Fst(QQ h))) f),
                         (H k2 k ([h:(toSet k)] (Snd(QQ h))) f))
  | (TO k1 k2)       => [QQ:((toSet k) -> ((toSet k1) -> (toSet k2)))]
                         ([g:(toSet k1)]
                            ( H k2 k ([h:(toSet k)] ((QQ h)(g))) f ))
  end) Q).


Definition K: (k:Kind)( ((toSet k)->(toSet k))
                      -> ((toSet k)->Omega) -> Omega) :=
  [k:Kind; f:((toSet k)->(toSet k)); g:((toSet k)->Omega)]
  (g(f(H k k ([x:(toSet k)](x)) f))).
```

Fig. 5. Coq code illustrating λ_i^{R}'s type system

D.4 Soundness

Lemma 20 (Substitution). *If* $\Gamma; \Delta \vdash e' : \tau'$ *and* $\Gamma, x : \tau'; \Delta \vdash e : \tau$, *then* $\Gamma; \Delta \vdash e[x := e'] : \tau$.

Proof. By induction on the derivation of $\Gamma, x : \tau'; \Delta \vdash e : \tau$. $\square$

Lemma 21 (Type Inversion).

1. *If* $\Gamma; \Delta \vdash x : \tau$, *then* $\Gamma(x) \equiv \tau : \Omega$.
2. *If* $\Gamma; \Delta \vdash n : \tau$, *then* $\tau \equiv \mathsf{int} : \Omega$.
3. *If* $\Gamma; \Delta \vdash op(e_1, e_2) : \tau$, *then* $\tau \equiv \mathsf{int} : \Omega$, $\Gamma; \Delta \vdash e_1 : \mathsf{int}$, *and* $\Gamma; \Delta \vdash e_2 : \mathsf{int}$.
4. *If* $\Gamma; \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau$, *then* $\Gamma; \Delta \vdash e : \mathsf{int}$, $\Gamma; \Delta \vdash e_1 : \tau$, *and* $\Gamma; \Delta \vdash e_2 : \tau$.
5. *If* $\Gamma; \Delta \vdash \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} : \tau$, *then* $\Gamma, f : \tau_1 \to \tau_2, x : \tau_1; \Delta \vdash e : \tau_2$ *and* $\tau \equiv \tau_1 \to \tau_2 : \Omega$.
6. *If* $\Gamma; \Delta \vdash e_1(e_2) : \tau$, *then there exists* τ_2 *such that* $\Gamma; \Delta \vdash e_1 : \tau_2 \to \tau$ *and* $\Gamma; \Delta \vdash e_2 : \tau_2$.
7. *If* $\Gamma; \Delta \vdash (e_1, e_2) : \tau$, *then there exist* τ_1 *and* τ_2 *such that* $\tau \equiv \tau_1 * \tau_2 : \Omega$, $\Gamma; \Delta \vdash e_1 : \tau_1$, *and* $\Gamma; \Delta \vdash e_2 : \tau_2$.
8. *If* $\Gamma; \Delta \vdash () : \tau$, *then* $\tau \equiv \mathsf{unit} : \Omega$.
9. *If* $\Gamma; \Delta \vdash \#i(e) : \tau$, *where* $i \in \{1, 2\}$, *then* $\Gamma; \Delta \vdash \tau_1 * \tau_2$ *and* $\tau \equiv \tau_i : \Omega$.
10. *If* $\Gamma; \Delta \vdash \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(e_i) : \tau$, *then* $\tau \equiv \tau_1 + \tau_2 : \Omega$ *and* $\Gamma; \Delta \vdash e_i : \tau_i$.
11. *If* $\Gamma; \Delta \vdash \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2\ \mathsf{end} : \tau$, *then* $\Gamma; \Delta \vdash e : \tau_1 + \tau_2$, $\Gamma, x_1 : \tau_1; \Delta \vdash e_1 : \tau$, *and* $\Gamma, x_2 : \tau_2; \Delta \vdash e_2 : \tau$.
12. *If* $\Gamma; \Delta \vdash \mathsf{roll}_{\mu_{\mathtt{toSet}(\kappa)}(f,g)}(e) : \tau$, *then* $\tau \equiv \mu_{\mathtt{toSet}(\kappa)}(f, g) : \Omega$ *and* $\Gamma; \Delta \vdash e : K(\kappa\, f\, g)$.
13. *If* $\Gamma; \Delta \vdash \mathsf{unroll}(e) : \tau$, *then there exist* f, g, *and* κ *such that* $\tau \equiv K(\kappa\, f\ g) : \Omega$, *and* $\Gamma; \Delta \vdash e : \mu_{\mathtt{toSet}(\kappa)}(f, g)$.

Proof (sketch). The proof relies on the fact that every typing derivation $\Gamma; \Delta \vdash e : \tau$ is the result of a unique typing rule, apart from the type-equality rule D.11. Since the type-equality rule has only one premise, each typing derivation ending with $\Gamma; \Delta \vdash e : \tau$ is a finite chain of zero or more type-equality judgments, with a necessarily unique type judgment at the top. $\square$

Theorem 22 (Preservation). *If* $\Gamma; \Delta \vdash e : \tau$ *and* $e \hookrightarrow e'$, *then* $\Gamma; \Delta \vdash e' : \tau$.

Proof. The proof is by induction on the rules defining one-step evaluation.

(Rule D.1) Here $e = op(v_1, v_2)$ and $e' = v_1\ op\ v_2$. By type inversion rule 3, $\tau \equiv \mathsf{int} : \Omega$, $\Gamma; \Delta \vdash v_1 : \mathsf{int}$, and $\Gamma; \Delta \vdash v_2 : \mathsf{int}$. The meta-level integer operation "op" returns an integer, so $\Gamma; \Delta \vdash e' : \mathsf{int}$.

(Rule D.2) Here $e = v_1(v_2)$, where $v_1 = \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e_1\ \mathsf{end}$. From syntax and type inversion rule 6, we know that $\Gamma; \Delta \vdash e : \tau_2$, and that $\Gamma; \Delta \vdash v_1 : \tau_1 \to \tau_2$. Using inversion again (rule 5), we have that $\Gamma, f : \tau_1 \to \tau_2, x : \tau_1; \Delta \vdash e_1 : \tau_2$. By twice applying the substitution lemma (20), we have that $\Gamma; \Delta \vdash e_1[f := v_1, x := v_2] : \tau_2$, which is what is required.

(Rule D.5) Here $e = \#i(v_1, v_2)$. By type inversion rule 9, we have $\Gamma; \Delta \vdash (v_1, v_2) : \tau_1 * \tau_2$ and $\Gamma; \Delta \vdash e : \tau_i$ for some τ_1 and τ_2. By type inversion rule 7, we have $\Gamma; \Delta \vdash v_i : \tau_i$, which is what is required.

(Rule D.6) Here $e = \mathsf{case}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2\ \mathsf{end}$, where $v = \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(v')$. From type inversion rule 11, we learn that $\Gamma; \Delta \vdash v : \tau_1 + \tau_2$ and $\Gamma, x : \tau_i; \Delta \vdash e_i : \tau$. Applying inversion again (rule 10) gives us $\Gamma; \Delta \vdash v' : \tau_i$. Using the substitution lemma (20), we have that $\Gamma; \Delta \vdash e_i[x := v'] : \tau$.

(Rule D.7) Here $e = \mathsf{unroll}(\mathsf{roll}_{\mu_{\mathtt{toSet}(\kappa)}(f,g)}(v))$ and $e' = v$. By type inversion rule 13, we have that $\Gamma; \Delta \vdash \mathsf{roll}_{\mu_{\mathtt{toSet}(\kappa)}(f,g)}(v) : \mu_{\mathtt{toSet}(\kappa)}(f, g)$ and that $\tau \equiv (K\ \kappa\ f\ g) : \Omega$. By type inversion rule 12, we have that $v : (K\ \kappa\ f\ g)$, and therefore from type equality we have $v : \tau$.

Proposition 23 (Canonical Forms). *Suppose that $v : \tau$ is a closed, well-formed value.*

1. *If $\vdash v : \mathit{int}$, then $v = n$ for some n.*
2. *If $\vdash v : \mathit{unit}$, then $v = ()$.*
3. *If $\vdash v : \tau_1 \rightarrow \tau_2$, then $v = \mathit{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end}$ for some f, x, and e.*
4. *If $\vdash v : \tau_1 * \tau_2$, then $v = (v_1, v_2)$ for some v_1 and v_2.*
5. *If $\vdash v : \tau_1 + \tau_2$, then $v = \mathit{inj}^{(i)}_{\tau_1+\tau_2}(v_i)$ for some v_i.*
6. *If $\vdash v : \mu_{\mathit{toSet}(\kappa)}(f, g)$, then $v = \mathit{roll}_{\mu_{\mathit{toSet}(\kappa)}(f,g)}(v')$ for some v'.*

Proof. The proof is by induction on the typing rules, using the fact that v is a value.

Theorem 24 (Progress). *If $\Gamma; \Delta \vdash e : \tau$, then either e is a value, or there exists e' such that $e \hookrightarrow e'$.*

Proof. The proof is by induction on the typing rules.

(Rule D.11) By application of the induction hypothesis.

(Rule D.12) Here n is already a value.

(Rule D.13) $\Gamma; \Delta \vdash op(e_1, e_2) : \mathsf{int}$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values. Thus by evaluation rule D.1, $\mathrm{op}(e_1, e_2) \hookrightarrow e_1\ \mathrm{op}\ e_2$.
- e_1 is not a value. Thus by the induction hypothesis, $e_1 \hookrightarrow e_1'$, and we apply the search rule.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2 \hookrightarrow e_2'$, and we apply the search rule.

(Rule D.14) $\Gamma; \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} : \tau$. There are two cases:

- e is a value. Using type inversion and canonical forms, either $e \mathrel{/=} 0$ or $e = 0$. In either case, $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \hookrightarrow e_i$, where $i = 1$ if e is nonzero or $i = 2$ otherwise.
- e is not a value. Then by the induction hypothesis, $e \hookrightarrow e'$, and thus we apply the search rule.

(Rule D.15) $\mathsf{fun}\ f(x:\tau_1):\tau_2\ \mathsf{is}\ e\ \mathsf{end}$ is a value.

(Rule D.16) $\Gamma;\Delta\vdash e_1(e_2):\tau_2$, where $\Gamma;\Delta\vdash e_1:\tau_1\to\tau_2$ and $\Gamma;\Delta\vdash e_2:\tau_1$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values. Using canonical forms, (Proposition 23), e_1 must be of the form $\mathsf{fun}\ f(x:\tau_1):\tau_2\ \mathsf{is}\ e'\ \mathsf{end}$. Using D.2, $e\hookrightarrow e'[f:=e_1,x:=e_2]$.
- e_1 is not a value. Thus by the induction hypothesis, $e_1\hookrightarrow e_1'$, and we apply the search rule.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2\hookrightarrow e_2'$, and we apply the search rule.

(Rule D.17) The expression $()$ is already a value.

(Rule D.18) $\Gamma;\Delta\vdash(e_1,e_2):\tau_1*\tau_2$, where $\Gamma;\Delta\vdash e_1:\tau_1$ and $\Gamma;\Delta\vdash e_2:\tau_2$. By the induction hypothesis, there are three cases:

- e_1 and e_2 are values; thus (e_1,e_2) is itself a value.
- e_1 is not a value. Thus by the induction hypothesis, $e_1\hookrightarrow e_1'$, and we apply the search rule.
- e_1 is a value, but e_2 is not. By the induction hypothesis, $e_2\hookrightarrow e_2'$, and we apply the search rule.

(Rule D.19) $\Gamma;\Delta\vdash \#i(e):\tau_i$, where $\Gamma;\Delta\vdash e:\tau_1*\tau_2$. By the induction hypothesis, there are two cases:

- e is a value. Thus by canonical forms (Proposition 23), e is of the form (v_1,v_2). Use evaluation rule D.5.
- e is not a value. By the induction hypothesis, then, $e\hookrightarrow e'$, and we apply the search rule.

(Rule D.20) $\Gamma;\Delta\vdash \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(e):\tau_1+\tau_2$, where $\Gamma;\Delta\vdash e:\tau_i$. By the induction hypothesis, there are two cases:

- e is a value; therefore $\mathsf{inj}^{(i)}_{\tau_1+\tau_2}(e)$ is itself a value.
- e is not a value. By the induction hypothesis, then, $e\hookrightarrow e'$, and we apply the search rule.

(Rule D.21) $\Gamma;\Delta\vdash \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inl}(x_1:\tau_1)\Rightarrow e_1\ \mathsf{orelse}\ \mathsf{inr}(x_2:\tau_2)\Rightarrow e_2\ \mathsf{end}:\tau$, where $\Gamma;\Delta\vdash e:\tau_1+\tau_2$, $\Gamma,x_1:\tau_1;\Delta\vdash e_1:\tau$, and $\Gamma,x_2:\tau_2;\Delta\vdash e_2:\tau$. By the induction hypothesis, there are two cases:

- e is a value. By canonical forms, $e=\mathsf{inj}^{(i)}_{\tau_1+\tau_2}(v_1)$. We apply D.6.
- e is not a value. By the induction hypothesis, then, $e\hookrightarrow e'$; use the search rule.

(Rule D.22) Here $e = \mathsf{unroll}(e_1)$, with $\Gamma; \Delta \vdash e_1 : \mu_{\mathtt{toSet}(\kappa)}(f, g)$. By the induction hypothesis, there are two cases:

- e_1 is a value. Thus, by canonical forms, $e_1 = \mathsf{roll}_{\mu_{\mathtt{toSet}(\kappa)}(f,g)}(v)$ for some v. Applying evaluation rule D.7 gives us $e \hookrightarrow v$.
- e_1 is not a value. Apply the search rule.

(Rule D.23) Here $e = \mathsf{roll}_{\mu_{\mathtt{toSet}(\kappa)}(f,g)}(e_1)$ with $e_1 : (K\,\kappa\,f\,g)$. By the induction hypothesis, there are two cases:

- e_1 is a value, therefore $e = \mathsf{roll}_{\mu_\kappa(f,g)}(e_1)$ is itself a value.
- e_1 is not a value. Apply the search rule.

D.5 Example: CPS Conversion (Details)

We start by defining a version of λ_i^R using type-annotated terms. By $\bar{f}$ and $\bar{e}$ we denote the terms without annotations. Type annotations allow us to present the CPS transformation based on syntax instead of typing derivations.

$$
\begin{array}{lll}
\text{(exp)} & e & ::= \bar{e}^A \\
& \bar{e} & ::= x \mid n \mid () \mid op(e_1, e_2) \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \\
& & \quad \mid \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \mathsf{is}\ e\ \mathsf{end} \mid e_1(e_2) \mid (e_1, e_2) \mid \#i(e) \\
& & \quad \mid \mathsf{case}_{\tau_1+\tau_2}\ e\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{or}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2 \\
& & \quad \mid \mathsf{inj}^{(i)}_{\tau_1+\tau_2}(e_i) \mid \mathsf{roll}_\tau(e) \mid \mathsf{unroll}(e) \\
\\
\text{(oper)} & op & ::= + \mid - \mid \ldots
\end{array}
$$

The target language $\lambda_{\mathcal{K}}$ of the CPS conversion stage has been defined in Section 6.4. We define the following syntactic sugar to denote non-recursive function definitions and value applications in $\lambda_{\mathcal{K}}$ (here x' is a fresh variable):

$$
\begin{aligned}
\lambda x : A.e &\equiv \mathsf{fix}\ x'[](x : A).e \\
v\,v' &\equiv v[](v') \\
\Lambda X_1 : A_1. \ldots \Lambda X_n.A_n.\lambda x : A.e &\equiv \mathsf{fix}\ x'[X_1 : A_1, \ldots, X_n : A_n](x : A).e
\end{aligned}
$$

In the static semantics of $\lambda_{\mathcal{K}}$ we use two forms of judgments. As in λ_i^R, the judgment $\Gamma; \Delta \vdash_K v : A$ indicates that the value v is well-formed with type A. In addition, the judgment $\Gamma; \Delta \vdash_K e$ indicates that the expression e is well-formed. We will omit the subscript on the $\vdash$ symbol when it can be deduced from the context.

The static semantics for $\lambda_{\mathcal{K}}$ is specified by the following rules (we specify only the rules which differ from those of λ_i^R):

$$
\frac{\begin{array}{c} A' = (\Pi X_1 : A_1. \ldots .\Pi X_n : A_n.A) \to \bot \\ \Delta \vdash A_i : s_i \ \text{ for all } i \in \{1, \ldots, n\} \\ \Delta, X_1 : A_1, \ldots, X_n : A_n \vdash A : \Omega_{\mathcal{K}} \\ \Gamma, x' : A', x : A; \Delta, X_1 : A_1, \ldots, X_n : A_n \vdash e \end{array}}{\Gamma; \Delta \vdash \mathsf{fix}\ x'[X_1 : A_1, \ldots, X_n : A_n](x : A).e : A'} \qquad \text{(D.24)}
$$

$$\frac{\begin{array}{c}\Delta \vdash A_i : B_i \text{ for all } i \in \{1, \dots, n\} \\ \Gamma; \Delta \vdash v' : (\Pi X_1 : A_1. \dots . \Pi X_n : A_n . A) \to \bot \\ \Gamma; \Delta \vdash v : [X_1 := A_1, \dots, X_n := A_n]A\end{array}}{\Gamma; \Delta \vdash v'[A_1, \dots, A_n](v)} \tag{D.25}$$

$$\frac{\Gamma; \Delta \vdash v : A \quad \Gamma, x : A; \Delta \vdash e}{\Gamma; \Delta \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e} \tag{D.26}$$

$$\frac{\Gamma; \Delta \vdash v : A_1 * A_2 \quad \Gamma, x : A_i; \Delta \vdash e}{\Gamma; \Delta \vdash \mathsf{let}\ x = \#i(v)\ \mathsf{in}\ e} \tag{D.27}$$

$$\frac{\Gamma; \Delta \vdash v_1 : \mathsf{int} \quad \Gamma; \Delta \vdash v_2 : \mathsf{int} \quad \Gamma, x : \mathsf{int} \vdash e}{\Gamma; \Delta \vdash \mathsf{let}\ x = v_1\ op\ v_2\ \mathsf{in}\ e} \tag{D.28}$$

$$\frac{\Gamma; \Delta \vdash v : \mu_{\mathtt{toSet}(\kappa)}(f, g) \quad \Gamma, x : (K\ \kappa\, f\, g); \Delta \vdash e}{\Gamma; \Delta \vdash \mathsf{let}\ x = \mathsf{unroll}(v)\ \mathsf{in}\ e} \tag{D.29}$$

$$\frac{\Gamma; \Delta \vdash v : \mathsf{int} \quad \Gamma; \Delta \vdash e_1 \quad \Gamma; \Delta \vdash e_2}{\Gamma; \Delta \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} \tag{D.30}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash v : \tau_1 + \tau_2 \\ \Gamma, x : \tau_1 \vdash e_1 \\ \Gamma, x : \tau_2 \vdash e_2 \\ \Gamma, x : \tau \vdash e\end{array}}{\Gamma; \Delta \vdash \mathsf{let}\ x = \mathsf{case}_{\tau_1 + \tau_2} v\ \mathsf{of}\ \mathsf{inj}^{(1)}(x : \tau_1) \Rightarrow e_1\ \mathsf{or}\ \mathsf{inj}^{(2)}(x : \tau_2) \Rightarrow e_2\ \mathsf{in}\ e} \tag{D.31}$$

The definition of the CPS transformation for computation terms of λ_i^R to computation terms of $\lambda_{\mathcal{K}}$ is given in Figure 6.

Proposition 25 (Type correctness of CPS conversion). *If* $\cdot; \cdot \vdash e : A$*, then* $\cdot; \cdot \vdash_K K_{exp} [\![e^A]\!] : (K_c(A) \to \bot)$.

$$
\begin{aligned}
&\mathsf{K}_{\mathrm{exp}}\,\llbracket x^{A} \rrbracket && = \lambda k : \mathsf{K}_c(A).k(x)\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket n^{\mathsf{int}} \rrbracket && = \lambda k : \mathsf{K}_c(\mathsf{int}).k(n)\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket ()^{\mathsf{unit}} \rrbracket && = \lambda k : \mathsf{K}_c(\mathsf{unit}).k(())\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket op(e_1^{\mathsf{int}}, e_2^{\mathsf{int}})^{\mathsf{int}} \rrbracket && = \lambda k : \mathsf{K}_c(\mathsf{int}).\mathsf{K}_{\mathrm{exp}}\,\llbracket e_1^{\mathsf{int}} \rrbracket\ (\lambda v_1 : \mathsf{K}_{\mathrm{typ}}(\mathsf{int}).\\
& && \qquad \mathsf{K}_{\mathrm{exp}}\,\llbracket e_2^{\mathsf{int}} \rrbracket\ (\lambda v_2 : \mathsf{K}_{\mathrm{typ}}(\mathsf{int}).\\
& && \qquad\quad (\mathsf{let}\ x = op(v_1, v_2)\ \mathsf{in}\ k\,x)))\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket \mathsf{if}\ e^{\mathsf{int}}\ \mathsf{then}\ e_1^{A}\ \mathsf{else}\ e_2^{A}\ \mathsf{fi} \rrbracket && = \lambda k : \mathsf{K}_c(A).\mathsf{K}_{\mathrm{exp}}\,\llbracket e^{\mathsf{int}} \rrbracket\ (\lambda x : \mathsf{int}.\\
& && \qquad \mathsf{if}\ x\ \mathsf{then}\ \mathsf{K}_{\mathrm{exp}}\,\llbracket e_1^{A} \rrbracket\ k\ \mathsf{else}\ \mathsf{K}_{\mathrm{exp}}\,\llbracket e_2^{A} \rrbracket\ k\ \mathsf{fi})\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket \mathsf{fun}\ f(x : A) : B\ \mathsf{is} && = \lambda k : \mathsf{K}_c(A \to B).k(\mathsf{fix}\ f[](k : \mathsf{K}_c(A \to B)).\\
&\qquad\quad e^{B}\ \mathsf{end}^{A \to B} \rrbracket && \qquad k(\lambda y : \mathsf{K}_{\mathrm{typ}}(A) * \mathsf{K}_c(B).\\
& && \qquad\quad \mathsf{let}\ x = \#1(y)\ \mathsf{in}\\
& && \qquad\qquad \mathsf{let}\ k = \#2(y)\ \mathsf{in}\ \mathsf{K}_{\mathrm{exp}}\,\llbracket e^{B} \rrbracket\ k))\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket (e_1^{A \to B}(e_2^{A}))^{B} \rrbracket && = \lambda k : \mathsf{K}_c(B).\mathsf{K}_{\mathrm{exp}}\,\llbracket e_1^{A \to B} \rrbracket\\
& && \qquad (\lambda x_1 : \mathsf{K}_{\mathrm{typ}}(A \to B).\mathsf{K}_{\mathrm{exp}}\,\llbracket e_2^{A} \rrbracket\\
& && \qquad\quad (\lambda x_2 : \mathsf{K}_{\mathrm{typ}}(A).x_1\ (x_2, k)))\\
&\mathsf{K}_{\mathrm{exp}}\,\left\llbracket (e_1^{A_1}, e_2^{A_2})^{A_1 * A_2} \right\rrbracket && = \lambda k : \mathsf{K}_c(A_1 * A_2).\\
& && \qquad \mathsf{K}_{\mathrm{exp}}\,\left\llbracket e_1^{A_1} \right\rrbracket\ (\lambda x_1 : \mathsf{K}_{\mathrm{typ}}(A_1).\\
& && \qquad\quad \mathsf{K}_{\mathrm{exp}}\,\left\llbracket e_2^{A_2} \right\rrbracket\ (\lambda x_2 : \mathsf{K}_{\mathrm{typ}}(A_2).k(x_1, x_2)))\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket \#i(e^{A_1 * A_2})^{A_i} \rrbracket && = \lambda k : \mathsf{K}_c(A_i).\mathsf{K}_{\mathrm{exp}}\,\llbracket e^{A_1 * A_2} \rrbracket\\
& && \qquad (\lambda v : \mathsf{K}_{\mathrm{typ}}(A_1 * A_2).\mathsf{let}\ x = \#i(v)\ \mathsf{in}\ k\,x)\\
&\mathsf{K}_{\mathrm{exp}}\,\llbracket \mathsf{case}_{A+B}\ e^{A+B}\ \mathsf{of} && = \lambda k : \mathsf{K}_c(C).\mathsf{K}_{\mathrm{exp}}\,\llbracket e^{A+B} \rrbracket\ (\lambda v : \mathsf{K}_{\mathrm{typ}}(A+B).\\
&\qquad\quad \mathsf{inj}^{(1)}(x : A) \Rightarrow e_1^{C} && \qquad \mathsf{let}\ x = \mathsf{case}_{\mathsf{K}_{\mathrm{typ}}(A+B)}\ v\ \mathsf{of}\\
&\qquad\quad \mathsf{or}\ \mathsf{inj}^{(2)}(x : B) \Rightarrow e_2^{C} \rrbracket && \qquad\quad \mathsf{inj}^{(1)}(x : A_1) \Rightarrow \mathsf{K}_{\mathrm{exp}}\,\llbracket e_1^{C} \rrbracket\ k\\
& && \qquad\quad \mathsf{or}\ \mathsf{inj}^{(2)}(x : A_2) \Rightarrow \mathsf{K}_{\mathrm{exp}}\,\llbracket e_2^{C} \rrbracket\ k\ \mathsf{in}\ e)\\
&\mathsf{K}_{\mathrm{exp}}\,\left\llbracket \mathsf{inj}_{A_1+A_2}^{(i)}(e^{A_i})^{A_1+A_2} \right\rrbracket && = \lambda k : \mathsf{K}_c(A_1 + A_2).\mathsf{K}_{\mathrm{exp}}\,\llbracket e^{A_i} \rrbracket\\
& && \qquad (\lambda v : \mathsf{K}_{\mathrm{typ}}(A_i).k\ \mathsf{inj}^{(i)}_{\mathsf{K}_{\mathrm{typ}}(A_1+A_2)}(v))\\
&\mathsf{K}_{\mathrm{exp}}\,\left\llbracket \mathsf{roll}_{\mu_{\mathtt{toSet}(\kappa)}(f,g)}(e^{K\ \kappa\ f\ g}) \right\rrbracket && = \lambda k : \mathsf{K}_c(\mu_{\mathtt{toSet}(\kappa)}(f, g)).\mathsf{K}_{\mathrm{exp}}\,\llbracket e^{K\ \kappa\ f\ g} \rrbracket\\
& && \qquad (\lambda v : \mathsf{K}_{\mathrm{typ}}(K\ \kappa\, f\ g).\\
& && \qquad\quad k\ \mathsf{roll}_{\mathsf{K}_{\mathrm{typ}}(\mu_{\mathtt{toSet}(k)}(f,g))}(v))\\
&\mathsf{K}_{\mathrm{exp}}\,\left\llbracket \mathsf{unroll}(e^{\mu_{\mathtt{toSet}(\kappa)}(f,g)})^{K\ \kappa\ f\ g} \right\rrbracket && = \lambda k : \mathsf{K}_c(K\ \kappa\, f\ g).\mathsf{K}_{\mathrm{exp}}\,\llbracket e^{\mu_{\mathtt{toSet}(\kappa)}} \rrbracket\\
& && \qquad (\lambda v : \mathsf{K}_{\mathrm{typ}}(\mu_{\mathtt{toSet}(\kappa)}(f, g)).\\
& && \qquad\quad \mathsf{let}\ x = \mathsf{unroll}(v)\ \mathsf{in}\ k\,x)
\end{aligned}
$$

Fig. 6. CPS conversion: from λ_i^{R} to $\lambda_{\mathcal{K}}$.