A Type System for Certified Binaries*

Zhong Shao Bratin Saha Valery Trifonov Nikolaos Papaspyrou
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{shao, saha, trifonov, nickie}@cs.yale.edu

July 23, 2001

Abstract

A *certified binary* is a value together with a proof that the value satisfies a given specification. Existing compilers that generate certified code have focused on simple memory and control-flow safety rather than more advanced properties. In this paper, we present a general framework for explicitly representing complex propositions and proofs in typed intermediate and assembly languages. The new framework allows us to reason about certified programs that involve effects while still maintaining decidable typechecking. We show how to integrate an entire proof system (the calculus of inductive constructions) into a compiler intermediate language and how the intermediate language can undergo complex transformations (CPS and closure conversion) while preserving proofs represented in the type system. Our work provides a foundation for the process of automatically generating certified binaries in a type-theoretic framework.

1 Introduction

Proof-carrying code (PCC), as pioneered by Necula and Lee [29, 28], allows a code producer to provide a machine-language program to a host, along with a formal proof of its safety. The proof can be mechanically checked by the host; the producer need not be trusted because a valid proof is incontrovertible evidence of safety.

The PCC framework is general because it can be applied to certify arbitrary data objects with complex specifications [31, 1]. For example, the Foundational PCC system [2] can certify any property expressible in Church's higher-order logic. Harper *et al.* [19, 6] call all these proof-carrying constructs certified binaries (or deliverables [6]). A *certified binary* is a value (which can be a function, a data structure, or a combination of both) together with a proof that the value satisfies a given specification.

Unfortunately, little is known on how to construct or generate certified binaries. Existing certifying compilers [30, 8] have focused on simple memory and control-flow safety only. Typed intermediate languages [21] and typed assembly languages [27] are effective techniques for automatically generating certified code; however, none of these type systems can rival the expressiveness of the actual higher-order logic as used in some PCC systems [2].

In this paper, we present a type-theoretic framework for constructing, composing, and reasoning about certified binaries. Our plan is to use the *formulae-as-types* principle [23] to represent propositions and proofs in a general type system, and then to investigate their relationship with compiler intermediate and assembly languages. We show how to integrate an entire proof system (the calculus of inductive constructions [34, 10]) into an intermediate language, and how to define complex transformations (CPS and closure conversion) of programs in this language so that they preserve proofs represented in the type system. Our paper builds upon a large body of previous work in the logic and theorem-proving community (see Barendregt *et al.* [4, 3] for a good summary), and makes the following new contributions:

- We show how to design new typed intermediate languages that are capable of representing and manipulating propositions and proofs. In particular, we show how to maintain decidability of typechecking when reasoning about certified programs that involve effects. This is different from the work done in the logic community which focuses on strongly normalizing (primitive recursive) programs.
- We maintain a phase distinction between compile-time typechecking and run-time evaluation. This property is often lost in the presence of dependent types (which are necessary for representing proofs in predicate logic). We achieve this by never having the type language (see Section 3) dependent on the computation language (see Section 4). Proofs are instead always represented at the type level using dependent kinds.
- We show how to use propositions to express program invariants and how to use proofs to serve as static capabilities. Following Xi and Pfenning [44], we use singleton types [22] to support the necessary interaction between the type and computation languages. We can assign an accurate type to unchecked vector (or array) access (see Section 4.2). Xi and Pfenning [44] can achieve the same using constraint checking, but their system does not support arbitrary propositions and (explicit) proofs, so it is less general than ours.
- We use a single type language to typecheck different compiler intermediate languages. This is crucial because it is impractical to have separate proof libraries for each intermediate language. We achieve this by using inductive definitions to define all types used to classify computation terms. This in turn nicely fits our work on (fully reflexive) intensional type analysis [39] into a single system.
- We show how to perform CPS and closure conversion on our intermediate languages while still preserving proofs represented in the type system. Existing algorithms [27, 20, 25, 5] all require that the transformation be performed on the entire type language. This is impractical because proofs are large in size; transforming them can alter their meanings and break the sharing among different languages. We present new techniques that completely solve these problems (Sections 5–6).

*This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

- Our type language is a variant of the calculus of inductive constructions [34, 10]. Following Werner [41], we give rigorous proofs for its meta-theoretic properties (subject reduction, strong normalization, confluence, and consistency of the underlying logic). We also give the soundness proof for our sample computation language. See Sections 3–4, the appendix, and the companion technical report [37] for details.

As far as we know, our work is the first comprehensive study on how to incorporate higher-order predicate logic (with inductive terms and predicates) into typed intermediate languages. Our results are significant because they open up many new exciting possibilities in the area of type-based language design and compilation. The fact that we can internalize a very expressive logic into our type system means that formal reasoning traditionally done at the meta level can now be expressed inside the actual language itself. For example, much of the past work on program verification using Hoare-like logics may now be captured and made explicit in a typed intermediate language.

From the standpoint of type-based language design, recent work [21, 44, 12, 14, 40, 39] has produced many specialized, increasingly complex type systems, each with its own meta-theoretical proofs, yet it is unclear how they will fit together. We can hope to replace them with one very general type system whose meta theory is proved once and for all, and that allows the definition of specialized type operators via the general mechanism of inductive definitions. For example, inductive definitions subsume and generalize earlier systems on intensional type analysis [21, 13, 39].

We have started implementing our new type system in the FLINT compiler [35, 36], but making the implementation realistic still involves solving many remaining problems (*e.g.*, efficient proof representations). Nevertheless, we believe our current contributions constitute a significant step toward the goal of providing a practical end-to-end compiler that generates certified binaries.

2 Approach

Our main objectives are to design typed intermediate and low-level languages that can directly manipulate propositions and proofs, and then to use them to certify realistic programs. We want our type system to be simple but general; we also want to support complex transformations (CPS and closure conversion) that preserve proofs represented in the type system. In this section, we describe the main challenges involved in achieving these goals and give an high-level overview of our main techniques.

Before diving into the details, we first establish a few naming conventions that we will use in the rest of this paper. Typed intermediate languages are usually structured in the same way as typed λ-calculi. Figure 1 gives a fragment of a richly typed λ-calculus, organized into four levels: kind schema (*kscm*) u, kind κ, type τ, and expression (*exp*) e. If we ignore kind schema and other extensions, this is just the polymorphic λ-calculus F_ω [18].

We divide each typed intermediate language into a type sub-language and a computation sub-language. The type language contains the top three levels. Kind schemas classify kind terms while kinds classify type terms. We often say that a kind term κ has kind schema u, or a type term τ has kind κ. We assume all kinds used to classify type terms have kind schema Kind, and all types used to classify expressions have kind Ω. Both the function type $\tau_1 \to \tau_2$ and the polymorphic type $\forall t : \kappa.\, \tau$ have kind Ω. Following the tradition, we sometimes say "a kind κ" to imply that κ has kind schema Kind, "a type τ" to imply that τ has kind Ω, and "a type constructor τ" to imply that τ has kind "$\kappa \to \cdots \to \Omega$." Kind terms with other kind schemas, or type terms with other kinds are strictly referred as "kind terms" or "type terms."

THE TYPE LANGUAGE:

(*kscm*) $u ::= \mathsf{Kind} \mid \ldots$

(*kind*) $\kappa ::= \kappa_1 \to \kappa_2 \mid \Omega \mid \ldots$

(*type*) $\tau ::= t \mid \lambda t{:}\kappa.\, \tau \mid \tau_1\ \tau_2 \mid \tau_1 \to \tau_2 \mid \forall t{:}\kappa.\, \tau \mid \ldots$

THE COMPUTATION LANGUAGE:

(*exp*) $e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \Lambda t{:}\kappa.\, e \mid e[\tau] \mid \ldots$

Figure 1: Typed λ-calculi—a skeleton

The computation language contains just the lowest level which is where we write the actual program. This language will eventually be compiled into machine code. We often use names such as computation terms, computation values, and computation functions to refer to various constructs at this level.

2.1 Representing propositions and proofs

The first step is to represent propositions and proofs for a particular logic in a type-theoretic setting. The most established technique is to use the *formulae-as-types* principle (a.k.a. the Curry-Howard correspondence) [23] to map propositions and proofs into a typed λ-calculus. The essential idea, which is inspired by constructive logic, is to use types (of kind Ω) to represent propositions, and expressions to represent proofs. A proof of an implication $P \supset Q$ is a function object that yields a proof of proposition Q when applied to a proof of proposition P. A proof of a conjunction $P \wedge Q$ is a pair (e_1, e_2) such that e_1 is a proof of P and e_2 is a proof of Q. A proof of disjunction $P \vee Q$ is a pair (b, e)—a tagged union—where b is either 0 or 1 and if $b = 0$, then e is a proof of P; if $b = 1$ then e is a proof of Q. There is no proof for the false proposition. A proof of a universally quantified proposition $\forall x \in B.P(x)$ is a function that maps every element b of the domain B into a proof of $P(b)$ where P is a unary predicate on elements of B. Finally, a proof of an existentially quantified proposition $\exists x \in B.P(x)$ is a pair (b, e) where b is an element of B and e is a proof of $P(b)$.

Proof-checking in the logic now becomes typechecking in the corresponding typed λ-calculus. There has been a large body of work done along this line in the last 30 years; most type-based proof assistants are based on this fundamental principle. Barendregt *et al.* [4, 3] give a good survey on previous work in this area.

2.2 Representing certified binaries

Under the type-theoretic setting, a certified binary S is just a pair (v, e) that consists of:

- a value v of type τ where v could be a function, a data structure, or any combination of both;
- and a proof e of $P(v)$ where P is a unary predicate on elements of type τ.

Here e is just an expression with type $P(v)$. The predicate P is a dependent type constructor with kind $\tau \to \Omega$. The entire package S has a dependent strong-sum type $\Sigma x{:}\tau.P(x)$.

For example, suppose Nat is the domain for natural numbers and $Prime$ is a unary predicate that asserts an element of Nat as a prime number, we introduce a type nat representing Nat, and a type constructor prime (of kind $\mathsf{nat} \to \Omega$) representing $Prime$. We can build a certified prime-number package by pairing a value v

(a natural number) with a proof for the proposition $\mathsf{prime}(v)$; the resulting certified binary has type $\Sigma x\!:\!\mathsf{nat}.\ \mathsf{prime}(x)$.

Function values can be certified in the same way. Given a function f that takes a natural number and returns another one as the result (*i.e.*, f has type $\mathsf{nat} \to \mathsf{nat}$), in order to show that f always maps a prime to another prime, we need a proof for the following proposition:

$$\forall x{\in}Nat.\ Prime(x) \supset Prime(f(x))$$

In a typed setting, this universally quantified proposition is represented as a dependent product type:

$$\Pi x\!:\!\mathsf{nat}.\ \mathsf{prime}(x) \to \mathsf{prime}(f(x))$$

The resulting certified binary has type

$$\Sigma f\!:\!\mathsf{nat} \to \mathsf{nat}.\ \Pi x\!:\!\mathsf{nat}.\ \mathsf{prime}(x) \to \mathsf{prime}(f(x))$$

Here the type is not only dependent on values but also on function applications such as $f(x)$, so verifying a certified binary involves typechecking the proof which in turn requires evaluating the underlying function application.

2.3 The problems with dependent types

The above scheme unfortunately fails to work in the context of typed intermediate (or assembly) languages. There are at least four problems with dependent types; the third and fourth are present even in the general context.

First, real programs often involve effects such as assignment, I/O, or non-termination. Effects interact badly with dependent types. In our previous example, suppose the function f does not terminate on certain inputs; then clearly, typechecking—which could involve applying f—would become undecidable. It is possible to use the effect discipline [38] to force types to be dependent on pure computation only, but this does not work in some typed λ-calculi; for example, a "pure" term in Girard's λU [18] could still diverge.

Even if applying f does not involve any effects, we still have more serious problems. In a type-preserving compiler, the body of the function f has to be compiled down to typed low-level languages. A few compilers perform typed CPS conversion [27], but in the presence of dependent types, this is still an open problem [5]. Also, typechecking in low-level languages would now require performing the equivalent of β-reductions on the low-level (assembly) code; this is awkward and difficult to support cleanly.

Third, it is important to maintain a phase distinction between compile-time typechecking and run-time evaluation. Having dependent strong-sum and dependent product types makes it harder to preserve this property. It is also difficult to support first-class certified binaries.

Finally, it would be nice to support a notion of subset types [9, 32]. A certified binary of type $\Sigma x\!:\!\mathsf{nat}.\ \mathsf{prime}(x)$ contains a natural number v and a proof that v is a prime. However, in some cases, we just want v to belong to a subset type $\{x\!:\!\mathsf{nat} \mid \mathsf{prime}(x)\}$, *i.e.*, v is a prime number but the proof of this is not together with v; instead, it can be constructed from the current context.

2.4 Separating the type and computation languages

We solve these problems by making sure that our type language is never dependent on the computation language. Because the actual program (*i.e.*, the computation term) would have to be compiled down to assembly code in any case, it is a bad idea to treat it as part of types. This strong separation immediately gives us back the phase-distinction property.

To represent propositions and proofs, we lift everything one level up: we use kinds to represent propositions, and type terms to represent proofs. The domain Nat is now represented by a kind Nat; the predicate $Prime$ is represented by a dependent kind term Prime which maps a type term of kind Nat into a proposition. A proof for proposition $\mathsf{Prime}(n)$ certifies that the type term n is a prime number.

To maintain decidable typechecking, we insist that the type language is strongly normalizing and free of side effects. This is possible because the type language no longer depends on any runtime computation. Given a type-level function g of kind $\mathsf{Nat} \to \mathsf{Nat}$, we can certify that it always maps a prime to another prime by building a proof τ_p for the following proposition, now represented as a dependent product kind:

$$\Pi t\!:\!\mathsf{Nat}.\mathsf{Prime}(t) \to \mathsf{Prime}(g(t)).$$

Essentially, we circumvent the problems with dependent types by replacing them with dependent kinds and by lifting everything (in the proof language) one level up.

To reason about actual programs, we still have to connect terms in the type language with those in the computation language. We follow Xi and Pfenning [44] and use singleton types [22] to relate computation values to type terms. In the previous example, we introduce a singleton type constructor snat of kind $\mathsf{Nat} \to \Omega$. Given a type term n of kind Nat, if a computation value v has type $\mathsf{snat}(n)$, then v denotes the natural number represented by n.

A certified binary for a prime number now contains three parts: a type term n of kind Nat, a proof for the proposition $\mathsf{Prime}(n)$, and a computation value of type $\mathsf{snat}(n)$. We can pack it up into an existential package and make it a first-class value with type:

$$\exists n\!:\!\mathsf{Nat}.\exists t\!:\!\mathsf{Prime}(n).\mathsf{snat}(n).$$

Here we use $\exists$ rather than Σ to emphasize that types and kinds are no longer dependent on computation terms. Under the erasure semantics [15], this certified binary is just an integer value of type $\mathsf{snat}(n)$ at run time.

A value v of the subset type (for prime numbers) would simply have type $\mathsf{snat}(n)$ as long as we can construct a proof for $\mathsf{Prime}(n)$ based on the information from the context.

We can also build certified binaries for programs that involve effects. Returning to our example, assume again that f is a function in the computation language which may not terminate on some inputs. Suppose we want to certify that if the input to f is a prime, and the call to f does return, then the result is also a prime. We can achieve this in two steps. First, we construct a type-level function g of kind $\mathsf{Nat} \to \mathsf{Nat}$ to simulate the behavior of f (on all inputs where f does terminate) and show that f has the following type:

$$\forall n\!:\!\mathsf{Nat}.\ \mathsf{snat}(n) \to \mathsf{snat}(g(n))$$

Here following Figure 1, we use $\forall$ and $\to$ to denote the polymorphic and function types for the computation language. The type for f says that if it takes an integer of type $\mathsf{snat}(n)$ as input and does not loop forever, then it will return an integer of type $\mathsf{snat}(g(n))$. Second, we construct a proof τ_p showing that g always maps a prime to another prime. The certified binary for f now also contains three parts: the type-level function g, the proof τ_p, and the computation function f itself. We can pack it into an existential package with type:

$$\exists g\!:\!\mathsf{Nat} \to \mathsf{Nat}.\ \exists p\!:\!(\Pi t\!:\!\mathsf{Nat}.\mathsf{Prime}(t) \to \mathsf{Prime}(g(t))).\ \forall n\!:\!\mathsf{Nat}.\ \mathsf{snat}(n) \to \mathsf{snat}(g(n))$$

Notice this type also contains function applications such as $g(n)$, but g is a type-level function which is always strongly normalizing, so typechecking is still decidable.

2.5 Designing the type language

We can incorporate propositions and proofs into typed intermediate languages, but designing the actual type language is still a challenge. For decidable typechecking, the type language should not depend on the computation language and it must satisfy the usual meta-theoretical properties (*e.g.* strong normalization).

But the type language also has to fulfill its usual responsibilities. First, it must provide a set of types (of kind Ω) to classify the computation terms. A typical compiler intermediate language supports a large number of basic type constructors (*e.g.*, integer, array, record, tagged union, and function). These types may change their forms during compilation, so different intermediate languages may have different definitions of Ω; for example, a computation function at the source level may be turned into CPS-style, or later, to one whose arguments are machine registers [27]. We also want to support intensional type analysis [21] which is crucial for typechecking runtime services [26].

Our solution is to provide a general mechanism of inductive definitions in our type language and to define each such Ω as an inductive kind. This was made possible only recently [39] and it relies on the use of polymorphic kinds. Taking the type language in Figure 1 as an example, we add kind variables k and polymorphic kinds $\Pi k : u.\, \kappa$, and replace Ω and its associated type constructors with inductive definitions (not shown):

$$
\begin{array}{ll}
(\textit{kscm}) & u ::= \mathsf{Kind} \mid \ldots \\
(\textit{kind}) & \kappa ::= \kappa_1 \to \kappa_2 \mid k \mid \Pi k : u.\, \kappa \mid \ldots \\
(\textit{type}) & \tau ::= t \mid \lambda t : \kappa.\, \tau \mid \tau_1\, \tau_2 \mid \lambda k : u.\, \tau \mid \tau[\kappa] \mid \ldots
\end{array}
$$

At the type level, we add kind abstraction $\lambda k : u.\, \tau$ and kind application $\tau[\kappa]$. The kind Ω is now inductively defined as follows (see Sections 3–4 for more details):

$$
\begin{array}{lll}
\mathsf{Inductive}\ \Omega : \mathsf{Kind} & := \twoheadrightarrow : & \Omega \to \Omega \to \Omega \\
& \mid \forall \ : & \Pi k : \mathsf{Kind}.\, (k \to \Omega) \to \Omega \\
& \vdots &
\end{array}
$$

Here $\twoheadrightarrow$ and $\forall$ are two of the constructors (of Ω). The polymorphic type $\forall t : \kappa.\, \tau$ is now written as $\forall[\kappa]\,(\lambda t : \kappa.\, \tau)$; the function type $\tau_1 \to \tau_2$ is just $\twoheadrightarrow \tau_1\, \tau_2$.

Inductive definitions also greatly increase the programming power of our type language. We can introduce new data objects (*e.g.*, integers, lists) and define primitive recursive functions, all at the type level; these in turn are used to help model the behaviors of the computation terms.

To have the type language double up as a proof language for higher-order predicate logic, we add dependent product kind $\Pi t : \kappa_1.\, \kappa_2$, which subsumes the arrow kind $\kappa_1 \to \kappa_2$; we also add kind-level functions to represent predicates. Thus the type language naturally becomes the calculus of inductive constructions [34].

2.6 Proof-preserving compilation

Even with a proof system integrated into our intermediate languages, we still have to make sure that they can be CPS- and closure-converted down to low-level languages. These transformations should preserve proofs represented in the type system; in fact, they should not traverse the proofs at all since doing so is impractical with large proof libraries.

These challenges are non-trivial but the way we set up our type system makes it easier to solve them. First, because our type language does not depend on the computation language, we do not have the difficulties involved in CPS-converting dependently typed λ-calculi [5]. Second, all our intermediate languages share the same type language thus also the same proof library; this is possible because the Ω kind (and the associated types) for each intermediate language is just a regular inductive definition.

Finally, a type-preserving program transformation often requires translating the source types (of the source Ω kind) into the target types (of the target Ω kind). Existing CPS- and closure-conversion algorithms [27, 20, 25] all perform such translation at the meta-level; they have to go through every type term (thus every proof term in our setting) during the translation, because any type term may contain a sub-term which has the source Ω kind. In our framework, the fact that each Ω kind is inductively defined means that we can internalize and write the type-translation function inside our type language itself. This leads to elegant algorithms that do not traverse any proof terms but still preserve typing and proofs (see Sections 5–6 for details).

2.7 Putting it all together

A certifying compiler in our framework will have a series of intermediate languages, each corresponding to a particular stage in the compilation process; all will share the same type language. An intermediate language is now just the type language plus the corresponding computation terms, along with the inductive definition for the corresponding Ω kind. In the rest of this paper, we first give a formal definition of our type language (which will be named as TL from now on) in Section 3; we then present a sample computation language λ_H in Section 4; we show how λ_H can be CPS- and closure-converted into low-level languages in Sections 5–6; finally, we discuss related work and then conclude.

3 The Type Language TL

Our type language TL resembles the calculus of inductive constructions (CIC) implemented in the Coq proof assistant [24]. This is a great advantage because Coq is a very mature system and it has a large set of proof libraries which we can potentially reuse. For this paper, we decided not to directly use CIC as our type language for three reasons. First, CIC contains some features designed for program extraction [33] which are not required in our case (where proofs are only used as specifications for the computation terms). Second, as far as we know, there are still no formal studies covering the entire CIC language. Third, for theoretical purposes, we want to understand what are the most essential features for modeling certified binaries.

Motivations Following the discussion in Section 2.5, we organize TL into the following three levels:

$$
\begin{array}{ll}
(\textit{kscm}) & u ::= z \mid \Pi t : \kappa.\, u \mid \Pi k : u_1.\, u_2 \mid \mathsf{Kind} \\
(\textit{kind}) & \kappa ::= k \mid \lambda t : \kappa_1.\, \kappa_2 \mid \kappa[\tau] \mid \lambda k : u.\, \kappa \mid \kappa_1\, \kappa_2 \\
& \quad \mid \Pi t : \kappa_1.\, \kappa_2 \mid \Pi k : u.\, \kappa \mid \Pi z : \mathsf{Kscm}.\, \kappa \\
& \quad \mid \mathsf{Ind}(k : \mathsf{Kind})\{\vec{\kappa}\} \mid \mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa}\} \\
(\textit{type}) & \tau ::= t \mid \lambda t : \kappa.\, \tau \mid \tau_1\, \tau_2 \mid \lambda k : u.\, \tau \mid \tau[\kappa] \\
& \quad \mid \lambda z : \mathsf{Kscm}.\, \tau \mid \tau[u] \mid \mathsf{Ctor}\,(i, \kappa) \\
& \quad \mid \mathsf{Elim}[\kappa', \kappa](\tau')\{\vec{\tau}\}
\end{array}
$$

Here kind schemas (*kscm*) classify kind terms while kinds classify type terms. There are variables at all three levels: kind-schema variables z, kind variables k, and type variables t. We have an external constant Kscm classifying all the kind schemas; essentially, TL has an additional level above *kscm*, of which Kscm is the sole member.

A good way to comprehend TL is to look at its five Π constructs: there are three at the kind level and two at the kind-schema

level. We use a few examples to explain why each of them is necessary. Following the tradition, we use arrow terms (*e.g.*, $\kappa_1 \to \kappa_2$) as a syntactic sugar for the non-dependent Π terms (*e.g.*, $\Pi t\!:\!\kappa_1.\,\kappa_2$ is non-dependent if t does not occur free in κ_2).

- Kinds $\Pi t : \kappa_1.\,\kappa_2$ and $\kappa_1 \to \kappa_2$ are used to typecheck the type-level function $\lambda t : \kappa.\,\tau$ and its application form $\tau_1\,\tau_2$. Assuming Ω and Nat are inductive kinds (defined later) and Prime is a predicate with kind schema $\mathsf{Nat} \to \mathsf{Kind}$, we can write a type term such as $\lambda t : \Omega.\,t$ which has kind $\Omega \to \Omega$, a type-level arithmetic function such as plus which has kind $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$, or the universally quantified proposition in Section 2.2 which is represented as a kind $\Pi t\!:\!\mathsf{Nat}.\mathsf{Prime}(t) \to \mathsf{Prime}(g(t))$.

- Kinds $\Pi k : u.\,\kappa$ and $u \to \kappa$ are used to typecheck the type-level kind abstraction $\lambda k\!:\!u.\,\tau$ and its application form $\tau[\kappa]$. As mentioned in Section 2.5, this is needed to support intensional analysis of quantified types [39]. It can also be used to define logic connectives and constants, *e.g.*

$$\begin{array}{rcl} \mathsf{True} : \mathsf{Kind} & = & \Pi k\!:\!\mathsf{Kind}.\,k \to k \\ \mathsf{False} : \mathsf{Kind} & = & \Pi k\!:\!\mathsf{Kind}.\,k \end{array}$$

  True has the polymorphic identity as a proof:

$$\mathsf{id} : \mathsf{True} \quad = \quad \lambda k\!:\!\mathsf{Kind}.\,\lambda t\!:\!k.\,t$$

  but False is not inhabited (this is essentially the consistency property of TL which we will show later).

- Kind $\Pi z : \mathsf{Kscm}.\,\kappa$ is used to typecheck the type-level kind-schema abstraction $\lambda z : \mathsf{Kscm}.\,\tau$ and its application form $\tau[u]$. This is not in the core calculus of constructions [10]. We use it in the inductive definition of Ω (see Section 4) where both the $\forall_{\mathsf{Kscm}}$ and $\exists_{\mathsf{Kscm}}$ constructors have kind $\Pi z : \mathsf{Kscm}.\,(z \to \Omega) \to \Omega$. These two constructors in turn allow us to typecheck predicate-polymorphic computation terms, which occur fairly often since the closure-conversion phase turns all functions with free predicate variables (e.g, Prime) into predicate-polymorphic ones.

- Kind schemas $\Pi t : \kappa.\,u$ and $\kappa \to u$ are used to typecheck the kind-level type abstraction $\lambda t\!:\!\kappa_1.\,\kappa_2$ and its application form $\kappa[\tau]$. The predicate Prime has kind schema $\mathsf{Nat} \to \mathsf{Kind}$. A predicate with kind schema $\Pi t : \mathsf{Nat}.\,\mathsf{Prime}(t) \to \mathsf{Kind}$ is only applicable to prime numbers. We can also define *e.g.* a binary relation:

$$\mathsf{LT} \quad : \quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}$$

  so that $\mathsf{LT}\ t_1\ t_2$ is a proposition asserting that the natural number represented by t_1 is less than that of t_2.

- Kind schemas $\Pi k : u_1.\,u_2$ and $u_1 \to u_2$ are used to typecheck the kind-level function $\lambda k : u.\,\kappa$ and its application form $\kappa_1\,\kappa_2$. We use it to write higher-order predicates and logic connectives. For example, the logical negation operator can be written as follows:

$$\mathsf{Not} : \mathsf{Kind} \to \mathsf{Kind} \quad = \quad \lambda k\!:\!\mathsf{Kind}.\,(k \to \mathsf{False})$$

  The consistency of TL implies that a proposition and its negation cannot be both inhabited—otherwise applying the proof of the second to that of the first would yield a proof of False.

TL also provides a general mechanism of inductive definitions [34]. The term $\mathsf{Ind}(k : \mathsf{Kind})\{\vec{\kappa}\}$ introduces an inductive kind k containing a list of constructors whose kinds are specified by $\vec{\kappa}$. Here k must only occur "positively" inside each κ_i

$$\begin{array}{l}
\mathsf{Inductive\ Bool} : \mathsf{Kind} := \mathsf{true} : \mathsf{Bool} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{false} : \mathsf{Bool} \\[1ex]
\mathsf{Inductive\ Nat} : \mathsf{Kind} := \mathsf{zero} : \mathsf{Nat} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} \\[1ex]
\mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{plus}(\mathsf{zero}) \quad = \quad \lambda t\!:\!\mathsf{Nat}.\,t \\
\mathsf{plus}(\mathsf{succ}\ t) \quad = \quad \lambda t'\!:\!\mathsf{Nat}.\,\mathsf{succ}\ ((\mathsf{plus}\ t)\ t') \\[1ex]
\mathsf{ifez} : \mathsf{Nat} \to (\Pi k\!:\!\mathsf{Kind}.\,k \to (\mathsf{Nat} \to k) \to k) \\
\mathsf{ifez}(\mathsf{zero}) \quad = \quad \lambda k\!:\!\mathsf{Kind}.\,\lambda t_1\!:\!k.\,\lambda t_2\!:\!\mathsf{Nat} \to k.\,t_1 \\
\mathsf{ifez}(\mathsf{succ}\ t) \quad = \quad \lambda k\!:\!\mathsf{Kind}.\,\lambda t_1\!:\!k.\,\lambda t_2\!:\!\mathsf{Nat} \to k.\,t_2\ t \\[1ex]
\mathsf{le} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\
\mathsf{le}(\mathsf{zero}) \quad = \quad \lambda t\!:\!\mathsf{Nat}.\,\mathsf{true} \\
\mathsf{le}(\mathsf{succ}\ t) \quad = \quad \lambda t'\!:\!\mathsf{Nat}.\,\mathsf{ifez}\ t'\ \mathsf{Bool}\ \mathsf{false}\ (\mathsf{le}\ t) \\[1ex]
\mathsf{lt} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\
\mathsf{lt} \quad = \quad \lambda t\!:\!\mathsf{Nat}.\,\mathsf{le}\ (\mathsf{succ}\ t) \\[1ex]
\mathsf{Cond} : \mathsf{Bool} \to \mathsf{Kind} \to \mathsf{Kind} \to \mathsf{Kind} \\
\mathsf{Cond}(\mathsf{true}) \quad = \quad \lambda k_1\!:\!\mathsf{Kind}.\,\lambda k_2\!:\!\mathsf{Kind}.\,k_1 \\
\mathsf{Cond}(\mathsf{false}) \quad = \quad \lambda k_1\!:\!\mathsf{Kind}.\,\lambda k_2\!:\!\mathsf{Kind}.\,k_2
\end{array}$$

Figure 2: Examples of inductive definitions

(see Appendix A for the formal definition of positivity). The term $\mathsf{Ctor}\,(i, \kappa)$ refers to the i-th constructor in an inductive kind κ. For presentation, we will use a more friendly syntax in the rest of this paper. An inductive kind $I = \mathsf{Ind}(k\!:\!\mathsf{Kind})\{\vec{\kappa}\}$ will be written as:

$$\begin{array}{rl}
\mathsf{Inductive}\ I : \mathsf{Kind} := & \mathsf{c}_1 : [I/k]\kappa_1 \\
& \mid \mathsf{c}_2 : [I/k]\kappa_2 \\
& \vdots \\
& \mid \mathsf{c}_n : [I/k]\kappa_n
\end{array}$$

We give an explicit name c_i to each constructor, so c_i is just an abbreviation of $\mathsf{Ctor}\,(i, I)$. For simplicity, the current version of TL does not include parameterized inductive kinds, but supporting them is quite straightforward [41, 34].

TL provides two iterators to support primitive recursion on inductive kinds. The small elimination $\mathsf{Elim}[\kappa', \kappa](\tau')\{\vec{\tau}\}$ takes a type term τ' of inductive kind κ', performs the iterative operation specified by $\vec{\tau}$ (which contains a branch for each constructor of κ'), and returns a type term of kind $\kappa[\tau']$ as the result. The large elimination $\mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa}\}$ takes a type term τ of inductive kind κ', performs the iterative operation specified by $\vec{\kappa}$, and returns a kind term of kind schema u as the result. These iterators generalize the $\mathsf{Typerec}$ operator used in intensional type analysis [21, 13, 39].

Figure 2 gives a few examples of inductive definitions including the inductive kinds Bool and Nat and several type-level functions which we will use in Section 4. The small elimination for Nat takes the following form $\mathsf{Elim}[\mathsf{Nat}, \kappa](\tau')\{\tau_1; \tau_2\}$. Here, κ is a dependent kind with kind schema $\mathsf{Nat} \to \mathsf{Kind}$; τ' is the argument which has kind Nat. The term in the zero branch, τ_1, has kind $\kappa[\tau']$. The term in the succ branch, τ_2, has kind $\mathsf{Nat} \to \kappa[\tau'] \to \kappa[\tau']$. TL uses the ι-reduction to perform the iterator operation. For example, the two ι-reduction rules for Nat work as follows:

$$\begin{array}{l}
\mathsf{Elim}[\mathsf{Nat}, \kappa](\mathsf{zero})\{\tau_1; \tau_2\} \leadsto_\iota \tau_1 \\
\mathsf{Elim}[\mathsf{Nat}, \kappa](\mathsf{succ}\ \tau)\{\tau_1; \tau_2\} \leadsto_\iota \tau_2\ \tau\ (\mathsf{Elim}[\mathsf{Nat}, \kappa](\tau)\{\tau_1; \tau_2\})
\end{array}$$

The general ι-reduction rule is defined formally in Appendix A. In our examples, we take the liberty of using the pattern-matching

$$
\begin{array}{lll}
(\textit{sort}) & s & ::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{var}) & X & ::= z \mid k \mid t \\
(\textit{ptm}) & A, B & ::= s \mid X \mid \lambda X : A.\, B \mid A\ B \mid \Pi X : A.\, B \\
 & & \quad \mid \mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\} \mid \mathsf{Ctor}\,(i, A) \\
 & & \quad \mid \mathsf{Elim}[A', B'](A)\{\vec{B}\}
\end{array}
$$

Figure 3: Syntax of the type language TL

syntax (as in ML) to express the iterator operations, but they can be easily converted back to the Elim form.

In Figure 2, plus is a function which calculates the sum of two natural numbers. The function ifez behaves like a switch statement: if its argument is zero, it returns a function that selects the first branch; otherwise, the result takes the second branch and applies it to the predecessor of the argument. The function le evaluates to true if its first argument is less than or equal to the second. The function lt performs the less-than comparison.

The definition of function Cond, which implements a conditional with result at the kind level, uses large elimination on Bool. It has the form $\mathsf{Elim}[\mathsf{Bool}, u](\tau)\{\kappa_1; \kappa_2\}$, where τ is of kind Bool; both the true and false branches (κ_1 and κ_2) have kind schema u.

Formalization We want to give a formal semantics to TL and then reason about its meta-theoretical properties. But the five Π constructs have many redundancies, so in the rest of this paper, we will model TL as a pure type system (PTS) [3] extended with inductive definitions. Intuitively, instead of having a separate syntactical category for each level, we collapse all kind schemas u, kind terms κ, type terms τ, and the external constant Kscm into a single set of *pseudoterms* (*ptm*), denoted as A or B. Similar constructs can now share typing rules and reduction relations.

Figure 3 gives the syntax of TL, written in PTS style. There is now only one Π construct ($\Pi X : A.\, B$), one λ-abstraction ($\lambda X : A.\, B$), and one application form ($A\ B$); two iterators for inductive definitions are also merged into one ($\mathsf{Elim}[A', B'](A)\{\vec{B}\}$). We use X and Y to represent generic variables, but we will still use t, k, and z if the class of a variable is clear from the context.

TL has the following PTS specification which we will use to derive its typing rules:

$$
\begin{array}{lll}
\mathcal{S} & = & \mathsf{Kind}, \mathsf{Kscm}, \mathsf{Ext} \\
\mathcal{A} & = & \mathsf{Kind} : \mathsf{Kscm}, \mathsf{Kscm} : \mathsf{Ext} \\
\mathcal{R} & = & (\mathsf{Kind}, \mathsf{Kind}), (\mathsf{Kscm}, \mathsf{Kind}), (\mathsf{Ext}, \mathsf{Kind}) \\
 & & (\mathsf{Kind}, \mathsf{Kscm}), (\mathsf{Kscm}, \mathsf{Kscm})
\end{array}
$$

Here $\mathcal{S}$ contains the set of sorts used to denote universes. We have to add the constant Ext to support quantification over Kscm. Our names for the sorts reflect the fact we lifted everything one level up; they are related to other systems via the following table:

Systems	Notations		
TL	Kind	Kscm	Ext
Werner [41]	Set	Type	Ext
Coq/CIC [24]	Set,Prop	Type(0)	Type(1)
Barendregt [3]	$*$	$\Box$	$\triangle$

The axioms in the set $\mathcal{A}$ denote the relationship between different sorts; an axiom "$s_1 : s_2$" means that s_2 classifies s_1. The rules in the set $\mathcal{R}$ are used to define well-formed Π constructs, from which we can deduce the set of well-formed λ-definitions and applications. For example, the five rules for TL can be related to the five Π constructs through the following table:

PTS rules\ptm	$\Pi X : A.\, B$	$\lambda X : A.\, B$	$A\ B$
(Kind, Kind)	$\Pi t : \kappa_1.\, \kappa_2$	$\lambda t : \kappa.\, \tau$	$\tau_1\ \tau_2$
(Kscm, Kind)	$\Pi k : u.\, \kappa$	$\lambda k : u.\, \tau$	$\tau[\kappa]$
(Ext, Kind)	$\Pi z : \mathsf{Kscm}.\, \kappa$	$\lambda z : \mathsf{Kscm}.\, \tau$	$\tau[u]$
(Kind, Kscm)	$\Pi t : \kappa.\, u$	$\lambda t : \kappa_1.\, \kappa_2$	$\kappa[\tau]$
(Kscm, Kscm)	$\Pi k : u_1.\, u_2$	$\lambda k : u.\, \kappa$	$\kappa\ \kappa'$

We define a context Δ as a list of bindings from variables to pseudoterms:

$$(\textit{ctxt}) \quad \Delta ::= \cdot \mid \Delta, X : A$$

The typing judgment for the PTS-style TL now takes the form $\Delta \vdash A : A'$ meaning that within context Δ, the pseudoterm A is well-formed and has A' as its classifier. We can now write a single typing rule for all the Π constructs:

$$\frac{\Delta \vdash A : s_1 \quad \Delta, X : A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X : A.\, B : s_2} \qquad (\text{PROD})$$

Take the rule (Kind, Kscm) as an example. To build a well-formed term $\Pi X : A.\, B$, which will be a kind schema (because s_2 is Kscm), we need to show that A is a well-formed kind and B is a well-formed kind schema assuming X has kind A. We can also share the typing rules for all the λ-definitions and applications:

$$\frac{\Delta, X : A \vdash B : B' \qquad \Delta \vdash \Pi X : A.\, B' : s}{\Delta \vdash \lambda X : A.\, B : \Pi X : A.\, B'} \qquad (\text{FUN})$$

$$\frac{\Delta \vdash A : \Pi X : B'.\, A' \quad \Delta \vdash B : B'}{\Delta \vdash A\ B : [B/X]A'} \qquad (\text{APP})$$

The reduction relations can also be shared. TL supports the standard β- and η-reductions (denoted as $\rightsquigarrow_\beta$ and $\rightsquigarrow_\eta$) plus the previously mentioned ι-reduction (denoted as $\rightsquigarrow_\iota$) on inductive objects (see Appendix A). We use $\rhd_\beta$, $\rhd_\eta$, and $\rhd_\iota$ to denote the relations that correspond to the rewriting of subterms using the relations $\rightsquigarrow_\beta$, $\rightsquigarrow_\eta$, and $\rightsquigarrow_\iota$ respectively. We use $\rightsquigarrow$ and $\rhd$ for the unions of the above relations. We also write $=_{\beta\eta\iota}$ for the reflexive-symmetric-transitive closure of $\rhd$.

The complete typing rules for TL and the definitions of all the reduction relations are given in Appendix A. Following Werner [41] and Geuvers [16], we have shown that TL satisfies all the key meta-theoretic properties including subject reduction, strong normalization, Church-Rosser (and confluence), and consistency of the underlying logic. The detailed proofs for these properties are given in the companion technical report [37].

4 The Computation Language λ_H

The language of computations λ_H for our high-level certified intermediate format uses proofs, constructed in the type language, to verify propositions which ensure the runtime safety of the program. Furthermore, in comparison with other higher-order typed calculi, the types assigned to programs can be more refined, since program invariants expressible in higher-order predicate logic can be represented in our type language. These more precise types serve as more complete specifications of the behavior of program components, and thus allow the static verification of more programs.

One approach to presenting a language of computations is to encode its syntax and semantics in a proof system, with the benefit of obtaining machine-checkable proofs of its properties, *e.g.* type safety. This appears to be even more promising for a system with a type language like CIC, which is more expressive than higher-order predicate logic: The CIC proofs of some program properties, embedded as type terms in the program, may not be easily representable in meta-logical terms, thus it may be simpler to perform

$$
\begin{array}{llll}
(\textit{exp}) & e & ::= & x \mid \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid f \mid \mathsf{fix}\ x\!:\!A.\, f \mid e\ e' \mid e[A] \\
& & & \mid \langle X\!=\!A,\ e\!:\!A' \rangle \mid \mathsf{open}\ e\ \mathsf{as}\ \langle X,\ x\rangle\ \mathsf{in}\ e' \\
& & & \mid \langle e_0,\ \ldots\ e_{n-1}\rangle \mid \mathsf{sel}[A](e, e') \mid e\ \mathit{aop}\ e' \\
& & & \mid e\ \mathit{cop}\ e' \mid \mathsf{if}[A, A'](e,\ X_1.\, e_1,\ X_2.\, e_2) \\
& & & \text{where } n \in \mathbb{N} \\
(\textit{fun}) & f & ::= & \lambda x\!:\!A.\, e \mid \Lambda X\!:\!A.\, f \\
(\textit{arith}) & \mathit{aop} & ::= & + \mid \ldots \\
(\textit{cmp}) & \mathit{cop} & ::= & < \mid \ldots
\end{array}
$$

Figure 4: Syntax of the computation language λ_H.

all the reasoning in CIC. However our exposition of the language TL is focused on its use as a type language, and consequently it does not include all features of CIC. We therefore leave this possibility for future work, and give a standard meta-logical presentation instead; we address some of the issues related to adequacy in our discussion of type safety.

In this section we often use the unqualified "term" to refer to a computation term (expression) e, with syntax defined in Figure 4. Most of the constructs are borrowed from standard higher-order typed calculi. To simplify the exposition we only consider constants representing natural numbers ($\overline{n}$ is the value representing $n \in \mathbb{N}$) and boolean values (tt and ff). The term-level abstraction and application are standard; type abstractions and fixed points are restricted to function values, with the call-by-value semantics in mind and to simplify the CPS and closure conversions. The type variable bound by a type abstraction, as well as the one bound by the open construct for packages of existential type, can have either a kind or a kind schema. Dually, the type argument in a type application, and the witness type term A in the package construction $\langle X\!=\!A,\ e\!:\!A' \rangle$ can be either a type term or a kind term.

The constructs implementing tuple operations, arithmetic, and comparisons have nonstandard static semantics, on which we focus in section 4.1, but their runtime behavior is standard. The branching construct is parameterized at the type level with a proposition (which is dependent on the value of the test term) and its proof; the proof is passed to the executed branch.

Dynamic semantics We present a small step call-by-value operational semantics for λ_H in the style of Wright and Felleisen [42]. The values are defined as

$$v ::= \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid f \mid \mathsf{fix}\ x\!:\!A.\, f \mid \langle X\!=\!A,\ v\!:\!A' \rangle \mid \langle v_0,\ \ldots\ v_{n-1}\rangle$$

The reduction relation $\hookrightarrow$ is specified by the rules

$$(\lambda x\!:\!A.\, e)\ v \ \hookrightarrow\ [v/x]e \qquad \text{(R-}\beta\text{)}$$

$$(\Lambda X\!:\!B.\, f)[A] \ \hookrightarrow\ [A/X]f \qquad \text{(R-TY-}\beta\text{)}$$

$$\mathsf{sel}[A](\langle v_0,\ \ldots\ v_{n-1}\rangle, \overline{m}) \ \hookrightarrow\ v_m \qquad (m < n) \qquad \text{(R-SEL)}$$

$$\begin{array}{l}\mathsf{open}\ \langle X'\!=\!A,\ v\!:\!A' \rangle\ \mathsf{as}\ \langle X,\ x\rangle\ \mathsf{in}\ e \\ \quad \hookrightarrow\ [v/x][A/X]e \end{array} \qquad \text{(R-OPEN)}$$

$$(\mathsf{fix}\ x\!:\!A.\, f)\ v \ \hookrightarrow\ ([\mathsf{fix}\ x\!:\!A.\, f/x]f)\ v \qquad \text{(R-FIX)}$$

$$(\mathsf{fix}\ x\!:\!A.\, f)[A'] \ \hookrightarrow\ ([\mathsf{fix}\ x\!:\!A.\, f/x]f)[A'] \qquad \text{(R-TYFIX)}$$

$$\overline{m} + \overline{n} \ \hookrightarrow\ \overline{m+n} \qquad \text{(R-ADD)}$$

$$\overline{m} < \overline{n} \ \hookrightarrow\ \mathsf{tt} \qquad (m < n) \qquad \text{(R-LT-T)}$$

$$\overline{m} < \overline{n} \ \hookrightarrow\ \mathsf{ff} \qquad (m \geq n) \qquad \text{(R-LT-F)}$$

$$\mathsf{if}[B, A](\mathsf{tt},\ X_1.\, e_1,\ X_2.\, e_2) \ \hookrightarrow\ [A/X_1]e_1 \qquad \text{(R-IF-T)}$$

$$\mathsf{if}[B, A](\mathsf{ff},\ X_1.\, e_1,\ X_2.\, e_2) \ \hookrightarrow\ [A/X_2]e_2 \qquad \text{(R-IF-F)}$$

An evaluation context E encodes the call-by-value discipline:

$$
\begin{array}{l}
E ::= \bullet \mid E\ e \mid v\ E \mid E[A] \mid \langle X\!=\!A,\ E\!:\!A' \rangle \\
\quad \mid \mathsf{open}\ E\ \mathsf{as}\ \langle X,\ x\rangle\ \mathsf{in}\ e \mid \mathsf{open}\ v\ \mathsf{as}\ \langle X,\ x\rangle\ \mathsf{in}\ E \\
\quad \mid \langle v_0,\ \ldots v_i,\ E,\ e_{i+2},\ \ldots,\ e_{n-1}\rangle \mid \mathsf{sel}[A](E, e) \\
\quad \mid \mathsf{sel}[A](v, E) \mid E\ \mathit{aop}\ e \mid v\ \mathit{aop}\ E \mid E\ \mathit{cop}\ e \\
\quad \mid v\ \mathit{cop}\ E \mid \mathsf{if}[A, A'](E,\ X_1.\, e_1,\ X_2.\, e_2)
\end{array}
$$

The notation $E\{e\}$ stands for the term obtained by replacing the hole $\bullet$ in E by e. The single step computation $\mapsto$ relates $E\{e\}$ to $E\{e'\}$ when $e \hookrightarrow e'$, and $\mapsto^*$ is its reflexive transitive closure.

As shown the semantics is standard except for some additional passing of type terms in R-SEL and R-IF-T/F. However an inspection of the rules shows that types are irrelevant for the evaluation, hence a type-erasure semantics, in which all type-related operations and parameters are erased, would be entirely standard.

4.1 Static semantics

The static semantics of λ_H shows the benefits of using a type language as expressive as TL. We can now define the type constructors of λ_H as constructors of an inductive kind Ω, instead of having them built into λ_H. As we will show in Section 5, this property is crucial for the conversion to CPS, since it makes possible transforming direct-style types to CPS types within the type language.

$$
\begin{array}{lll}
\mathsf{Inductive}\ \Omega : \mathsf{Kind} & := \mathsf{snat} & : \mathsf{Nat} \to \Omega \\
& \mid \mathsf{sbool} & : \mathsf{Bool} \to \Omega \\
& \mid \twoheadrightarrow & : \Omega \to \Omega \to \Omega \\
& \mid \mathsf{tup} & : \mathsf{Nat} \to (\mathsf{Nat} \to \Omega) \to \Omega \\
& \mid \forall_{\mathsf{Kind}} & : \Pi k\!:\!\mathsf{Kind}.\, (k \to \Omega) \to \Omega \\
& \mid \exists_{\mathsf{Kind}} & : \Pi k\!:\!\mathsf{Kind}.\, (k \to \Omega) \to \Omega \\
& \mid \forall_{\mathsf{Kscm}} & : \Pi z\!:\!\mathsf{Kscm}.\, (z \to \Omega) \to \Omega \\
& \mid \exists_{\mathsf{Kscm}} & : \Pi z\!:\!\mathsf{Kscm}.\, (z \to \Omega) \to \Omega
\end{array}
$$

Informally, all well-formed computations have types of kind Ω, including singleton types of natural numbers $\mathsf{snat}\ A$ and boolean values $\mathsf{sbool}\ B$, as well as function, tuple, polymorphic and existential types. To improve readability we also define the syntactic sugar

$$
\begin{array}{l}
A \to B \equiv \twoheadrightarrow\ A\ B \\
\left.\begin{array}{l} \forall_s X\!:\!A.\, B \equiv \forall_s\ A\ (\lambda X\!:\!A.\, B) \\ \exists_s X\!:\!A.\, B \equiv \exists_s\ A\ (\lambda X\!:\!A.\, B) \end{array}\right\} \text{where } s \in \{\mathsf{Kind}, \mathsf{Kscm}\}
\end{array}
$$

and often drop the sort s when $s = \mathsf{Kind}$; *e.g.* the type void, containing no values, is defined as $\forall t\!:\!\Omega.\, t \equiv \forall_{\mathsf{Kind}}\ \Omega\ (\lambda t\!:\!\Omega.\, t)$.

Using this syntactic sugar we can give a familiar look to many of the formation rules for λ_H expressions and functional values. Figure 5 contains the inference rules for deriving judgments of the form $\Delta;\ \Gamma\ \vdash\ e : A$, which assign type A to the expression e in a context Δ and a type environment Γ defined by

$$(\textit{type env}) \quad \Gamma ::= \cdot \mid \Gamma, x\!:\!A$$

We introduce some of the notation used in these rules in the course of the discussion.

Rules E-NAT, E-TRUE, and E-FALSE assign singleton types to numeric and boolean constants. For instance the constant $\overline{1}$ has type $\mathsf{succ}\ \mathsf{zero}$ in any valid environment. In rule E-NAT we use the meta-function $\widehat{\cdot}$ to map natural numbers $n \in \mathbb{N}$ to their representations as type terms. It is defined inductively by $\widehat{0} = \mathsf{zero}$ and $\widehat{n+1} = \mathsf{succ}\ \widehat{n}$, so $\Delta\ \vdash\ \widehat{n} : \mathsf{Nat}$ holds for all valid Δ and $n \in \mathbb{N}$.

Singleton types play a central role in reflecting properties of values in the type language, where we can reason about them constructively. For instance rules E-ADD and E-LT use respectively the

$$\frac{\Delta \vdash \mathsf{Kind} : \mathsf{Kscm}}{\Delta \vdash \cdot \ ok} \quad \text{(TE-MT)}$$

$$\frac{\Delta \vdash \Gamma \ ok \qquad \Delta \vdash A : \Omega}{\Delta \vdash \Gamma, x{:}A \ ok} \quad \text{(TE-EXT)}$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash x : \Gamma(x)} \quad \text{(E-VAR)}$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \overline{n} : \mathsf{snat}\ \widehat{n}} \quad \text{(E-NAT)}$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \mathsf{tt} : \mathsf{sbool}\ \mathsf{true}} \quad \text{(E-TRUE)}$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \mathsf{ff} : \mathsf{sbool}\ \mathsf{false}} \quad \text{(E-FALSE)}$$

$$\frac{\Delta \vdash A : \Omega \qquad \Delta; \Gamma, x{:}A \vdash f : A}{\Delta; \Gamma \vdash \mathsf{fix}\ x{:}A.\, f : A} \quad \text{(E-FIX)}$$

$$\frac{\Delta \vdash A : \Omega \qquad \Delta; \Gamma, x{:}A \vdash e : A'}{\Delta; \Gamma \vdash \lambda x{:}A.\, e : A \to A'} \quad \text{(E-FUN)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : A \to A' \qquad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\, e_2 : A'} \quad \text{(E-APP)}$$

$$\frac{\Delta \vdash B : s \qquad \Delta, X{:}B; \Gamma \vdash f : A}{\Delta; \Gamma \vdash \Lambda X{:}B.\, f : \forall_s X{:}B.\, A} \ \begin{pmatrix} X \notin \Delta \\ s \neq \mathsf{Ext} \end{pmatrix} \quad \text{(E-TFUN)}$$

$$\frac{\Delta; \Gamma \vdash e : \forall_s X{:}B.\, A' \qquad \Delta \vdash A : B}{\Delta; \Gamma \vdash e[A] : [A/X]A'} \ (s \neq \mathsf{Ext}) \quad \text{(E-TAPP)}$$

$$\frac{\begin{array}{c}\Delta \vdash A : B \qquad \Delta \vdash B : s \\ \Delta; \Gamma \vdash e : [A/X]A'\end{array}}{\Delta; \Gamma \vdash \langle X{=}A,\, e{:}A' \rangle : \exists_s X{:}B.\, A'} \ (s \neq \mathsf{Ext}) \quad \text{(E-PACK)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \exists_s X'{:}B.\, A \qquad \Delta \vdash A' : \Omega \\ \Delta, X{:}B; \Gamma, x{:}[X/X']A \vdash e' : A'\end{array}}{\Delta; \Gamma \vdash \mathsf{open}\ e\ \mathsf{as}\ \langle X,\, x \rangle\ \mathsf{in}\ e' : A'} \ \begin{pmatrix} X \notin \Delta \\ s \neq \mathsf{Ext} \end{pmatrix} \quad \text{(E-OPEN)}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{snat}\ A \qquad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A'}{\Delta; \Gamma \vdash e + e' : \mathsf{snat}\ (\mathsf{plus}\ A\ A')} \quad \text{(E-ADD)}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{snat}\ A \qquad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A'}{\Delta; \Gamma \vdash e < e' : \mathsf{sbool}\ (\mathsf{lt}\ A\ A')} \quad \text{(E-LT)}$$

$$\frac{\begin{array}{c}\mathit{for\ all}\ i < n \qquad \Delta; \Gamma \vdash e_i : A_i \\ \Delta; \Gamma \vdash \langle e_0,\, \ldots\, e_{n-1} \rangle \\ \quad : \mathsf{tup}\ \widehat{n}\ (\mathsf{nth}\ (A_0{::}\ldots{::}A_{n-1}{::}\mathsf{nil}))\end{array}}{} \quad \text{(E-TUP)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \mathsf{tup}\ A''\ B \qquad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A' \\ \Delta \vdash A : \mathsf{LT}\ A'\ A''\end{array}}{\Delta; \Gamma \vdash \mathsf{sel}[A](e, e') : B\ A'} \quad \text{(E-SEL)}$$

$$\frac{\begin{array}{ll}\Delta \vdash B : \mathsf{Bool} \to \mathsf{Kind} & \Delta; \Gamma \vdash e : \mathsf{sbool}\ A'' \\ \Delta \vdash A : B\ A'' & \Delta, X_1{:}B\ \mathsf{true}; \Gamma \vdash e_1 : A' \\ \Delta \vdash A' : \Omega & \Delta, X_2{:}B\ \mathsf{false}; \Gamma \vdash e_2 : A'\end{array}}{\Delta; \Gamma \vdash \mathsf{if}[B, A](e,\, X_1.\, e_1,\, X_2.\, e_2) : A'} \quad \text{(E-IF)}$$

$$\frac{\Delta; \Gamma \vdash e : A \qquad A =_{\beta\eta\iota} A' \qquad \Delta \vdash A' : \Omega}{\Delta; \Gamma \vdash e : A'} \quad \text{(E-CONV)}$$

Figure 5: Static semantics of the computation language λ_H.

type terms plus and lt (defined in Section 3) to reflect the semantics of the term operations into the type level via singleton types.

However, if we could only assign singleton types to computation terms, in a decidable type system we would only be able to typecheck terminating programs. We regain expressiveness of the computation language using existential types to hide some of the too detailed type information. Thus for example one can define the usual types of all natural numbers and boolean values as

$$\begin{array}{l}\mathsf{nat}\ : \Omega = \exists t{:}\mathsf{Nat}.\, \mathsf{snat}\ t \\ \mathsf{bool} : \Omega = \exists t{:}\mathsf{Bool}.\, \mathsf{sbool}\ t\end{array}$$

For any term e with singleton type $\mathsf{snat}\ A$ the package $\langle t{=}A,\, e : \mathsf{snat}\ t \rangle$ has type nat. Since in a type-erasure semantics of λ_H all types and operations on them are erased, there is no runtime overhead for the packaging. For each $n \in \mathbb{N}$ there is a value of this type denoted by $\widehat{\overline{n}} \equiv \langle t = \widehat{n},\, \overline{n} : \mathsf{snat}\ t \rangle$. Operations on terms of type nat are derived from operations on terms of singleton types of the form $\mathsf{snat}\ A$; for example an addition function of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ is defined as the expression

$$\begin{array}{l}\mathsf{add} = \lambda \mathsf{x}_1{:}\mathsf{nat}.\, \lambda \mathsf{x}_2{:}\mathsf{nat}. \\ \qquad \mathsf{open}\ \mathsf{x}_1\ \mathsf{as}\ \langle t_1,\, \mathsf{x}_1' \rangle\ \mathsf{in}\ \mathsf{open}\ \mathsf{x}_2\ \mathsf{as}\ \langle t_2,\, \mathsf{x}_2' \rangle\ \mathsf{in} \\ \qquad\quad \langle t{=}\mathsf{plus}\ t_1\ t_2,\, \mathsf{x}_1' + \mathsf{x}_2' : \mathsf{snat}\ t \rangle\end{array}$$

Rule E-TUP assigns to a tuple a type of the form $\mathsf{tup}\ A\ B$, in which the tup constructor is applied to a type A representing the tuple size, and a function B mapping offsets to the types of the tuple components. This function is defined in terms of operations on lists of types:

$$\begin{array}{l}\mathsf{Inductive\ List}\ :\ \mathsf{Kind}\ := \mathsf{nil}\quad : \mathsf{List} \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{cons} : \Omega \to \mathsf{List} \to \mathsf{List}\end{array}$$

$$\begin{array}{ll}\mathsf{nth}\ :\ \mathsf{List} \to \mathsf{Nat} \to \Omega & \\ \mathsf{nth}\ \mathsf{nil} & = \lambda t{:}\mathsf{Nat}.\, \mathsf{void} \\ \mathsf{nth}\ (\mathsf{cons}\ t_1\ t_2) & = \lambda t{:}\mathsf{Nat}.\, \mathsf{ifez}\ t\ \Omega\ t_1\ (\mathsf{nth}\ t_2)\end{array}$$

Thus $\mathsf{nth}\ \mathsf{L}\ \widehat{n}$ reduces to the n-th element of the list L when n is less than the length of L, and to void otherwise. We also use the infix form $A{::}A' \equiv \mathsf{cons}\ A\ A'$. The type of pairs is derived: $A \times A' \equiv \mathsf{tup}\ \widehat{2}\ (\mathsf{nth}\ (A{::}A'{::}\mathsf{nil}))$. Thus for instance $\cdot;\cdot \vdash \langle \overline{42}, \overline{7} \rangle : \mathsf{snat}\ \widehat{42} \times \mathsf{snat}\ \widehat{7}$ is a valid judgment.

The rules for selection and testing for the less-than relation (the only comparison we discuss for brevity) refer to the kind term LT with kind schema $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}$. Intuitively, LT represents a binary relation on kind Nat, so $\mathsf{LT}\ \widehat{m}\ \widehat{n}$ is the kind of type terms representing proofs of $m < n$. LT can be thought of as the parameterized inductive kind of proofs constructed from instances of the axioms $\forall n \in \mathbb{N}.\, 0 < n{+}1$ and $\forall m, n \in \mathbb{N}.\, m < n \supset m{+}1 < n{+}1$:

$$\begin{array}{l}\mathsf{Inductive\ LT}\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind} \\ \quad := \mathsf{ltzs} : \Pi t{:}\mathsf{Nat}.\, \mathsf{LT}\ \mathsf{zero}\ (\mathsf{succ}\ t) \\ \qquad \mid \mathsf{ltss} : \Pi t{:}\mathsf{Nat}.\, \Pi t'{:}\mathsf{Nat}.\, \mathsf{LT}\ t\ t' \to \mathsf{LT}\ (\mathsf{succ}\ t)\ (\mathsf{succ}\ t')\end{array}$$

To simplify the presentation of our type language, we allowed inductive kinds of kind scheme Kind only. Thus to stay within the scope of this paper we actually use a Church encoding of LT (see

Appendix C for details); this is sufficient since proof objects are never analyzed in λ_H, so the full power of elimination is not necessary for LT.

In the component selection construct $\mathsf{sel}[A](e, e')$ the type A represents a *proof* that the value of the subscript e' is less than the size of the tuple e. In rule E-SEL this condition is expressed as an application of the type term LT. Due to the consistency of the logic represented in the type language, only the existence and not the structure of the proof object A is important. Since its existence is ensured statically in a well-formed expression, A would be eliminated in a type-erasure semantics.

The branching construct $\mathsf{if}[B, A](e,\ X_1.\, e_1,\ X_2.\, e_2)$ takes a type term A representing a proof of the proposition encoded as either B true or B false, depending on the value of e. The proof is passed to the appropriate branch in its bound type variable (X_1 or X_2). The correspondence between the value of e and the kind of A is again established through a singleton type. Note that unlike Xi and Harper [43] we allow imprecise information flow into the branches by not restricting B false to be the negation of B true. In particular this makes possible the encoding of the usual oblivious (in proof-passing sense) if using $B = \lambda t : \mathsf{Bool}.\ \mathsf{True}$.

4.2 Example: bound check elimination

A simple example of the generation, propagation, and use of proofs in λ_H is a function which computes the sum of the components of any vector of naturals. Let us first introduce some auxiliary types and functions. The type assigned to a homogeneous tuple (vector) of n terms of type A is $\beta\eta\iota$-convertible to the form $\mathsf{vec}\ \widehat{n}\ A$ for

$$\begin{aligned}&\mathsf{vec}\ :\ \mathsf{Nat} \to \Omega \to \Omega\\ &\mathsf{vec} = \lambda t : \mathsf{Nat}.\ \lambda t' : \Omega.\ \mathsf{tup}\ t\ (\mathsf{nth}\ (\mathsf{repeat}\ t\ t'))\end{aligned}$$

where

$$\begin{aligned}&\mathsf{repeat}\ :\ \mathsf{Nat} \to \Omega \to \mathsf{List}\\ &\mathsf{repeat}\ \mathsf{zero} \quad = \lambda t' : \Omega.\ \mathsf{nil}\\ &\mathsf{repeat}\ (\mathsf{succ}\ t) = \lambda t' : \Omega.\ t' :: (\mathsf{repeat}\ t)\ t'\end{aligned}$$

Then we can define a term which sums the elements of a vector with a given length as follows:

$$\begin{aligned}&\mathsf{sumVec} : \forall t : \mathsf{Nat}.\ \mathsf{snat}\ t \to \mathsf{vec}\ t\ \mathsf{nat} \to \mathsf{nat}\\ &\quad \equiv \Lambda t : \mathsf{Nat}.\ \lambda \mathsf{n} : \mathsf{snat}\ t.\ \lambda \mathsf{v} : \mathsf{vec}\ t\ \mathsf{nat}.\\ &\qquad (\mathsf{fix}\ \mathsf{loop} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\\ &\qquad\ \ \lambda \mathsf{i} : \mathsf{nat}.\ \lambda \mathsf{sum} : \mathsf{nat}.\\ &\qquad\qquad \mathsf{open}\ \mathsf{i}\ \mathsf{as}\ \langle t',\ \mathsf{i}' \rangle\ \mathsf{in}\\ &\qquad\qquad\ \mathsf{if}[\mathsf{LTOrTrue}\ t'\ t, \mathsf{ltPrf}\ t'\ t]\\ &\qquad\qquad\quad (\mathsf{i}' < \mathsf{n},\\ &\qquad\qquad\qquad t_1.\ \mathsf{loop}\ (\mathsf{add}\ \mathsf{i}\ \widehat{1})\\ &\qquad\qquad\qquad\qquad\qquad (\mathsf{add}\ \mathsf{sum}\ (\mathsf{sel}[t_1](\mathsf{v}, \mathsf{i}'))),\\ &\qquad\qquad\qquad t_2.\ \mathsf{sum}))\ \widehat{0}\ \widehat{0}\end{aligned}$$

where

$$\begin{aligned}&\mathsf{LTOrTrue}\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \to \mathsf{Kind}\\ &\mathsf{LTOrTrue} = \lambda t_1 : \mathsf{Nat}.\ \lambda t_2 : \mathsf{Nat}.\ \lambda t : \mathsf{Bool}.\ \mathsf{Cond}\ t\ (\mathsf{LT}\, t_1\, t_2)\ \mathsf{True}\end{aligned}$$

and ltPrf of kind $\Pi t' : \mathsf{Nat}.\ \Pi t : \mathsf{Nat}.\ \mathsf{LTOrTrue}\ t'\ t\ (\mathsf{lt}\ t'\ t)$ is a type term defined in Appendix C.

The comparison $\mathsf{i}' < \mathsf{n}$, used in this example as a loop termination test, checks whether the index i' is smaller than the vector size n. If it is, the adequacy of the type term lt with respect to the less-than relation ensures that the type term $\mathsf{ltPrf}\ t'\ t$ represents a proof of the corresponding proposition at the type level, namely $\mathsf{LT}\ t'\ t$. This proof is then bound to t_1 in the first branch of the if, and the sel construct uses it to verify that the i'-th element of v exists, thus avoiding a second test. The type safety of λ_H (Theorem 1) guarantees that implementations of sel need not check the subscript at runtime. Since the proof t_2 is ignored in the "else" branch, $\mathsf{ltPrf}\ t'\ t$ is defined to reduce to the trivial proof of True when the value of i' is not less than that of n.

The usual vector type, which keeps the length packaged with the content, is

$$\mathsf{vector}\ :\ \Omega \to \Omega = \lambda t : \Omega.\ \exists t' : \mathsf{Nat}.\ \mathsf{snat}\ t' \times \mathsf{vec}\ t'\ t.$$

Now we can write a wrapper function for sumVec with the standard type $\mathsf{vector\ nat} \to \mathsf{nat}$; we leave the details to the reader.

4.3 Type safety

The type safety of λ_H is a corollary of its properties of progress and subject reduction. A pivoting element in proving progress (Lemma 4 in Appendix B) is the connection between the existence of a proof (type) term of kind $\mathsf{LT}\ \widehat{m}\ \widehat{n}$, provided by rule E-SEL, and the existence of a (meta-logical) proof of the side condition $m < n$, required by rule R-SEL. Similarly, subject reduction (Lemma 5 in Appendix B) in the cases of R-ADD and R-LT-T/F relies on the adequate representation of addition and comparison by plus and lt.

Lemma 1 (Adequacy of the TL representation of arithmetic)

1. For all $m, n \in \mathbb{N}$, $\mathsf{plus}\ \widehat{m}\ \widehat{n} =_{\beta\eta\iota} \widehat{m+n}$.
2. For all $m, n \in \mathbb{N}$, $\mathsf{lt}\ \widehat{m}\ \widehat{n} =_{\beta\eta\iota} \mathsf{true}$ if and only if $m < n$.
3. For all $m, n \in \mathbb{N}$, $m < n$ if and only if there exists a type A such that $\cdot \vdash A : \mathsf{LT}\ \widehat{m}\ \widehat{n}$.

Proof sketch (3) For the forward direction it suffices to observe that the structure of the meta-logical proof of $m < n$ (in terms of the above axioms of ordering) can be directly reflected in a type term of kind $\mathsf{LT}\ \widehat{m}\ \widehat{n}$. The inverse direction is shown by examining the structure of closed type terms of this kind in normal form. □

Theorem 1 (Safety of λ_H) If $\cdot;\cdot \vdash e : A$, then either $e \mapsto^* v$ and $\cdot;\cdot \vdash v : A$, or e diverges (*i.e.*, for each e', if $e \mapsto^* e'$, then there exists e'' such that $e' \mapsto e''$).

Proof sketch Follows from Lemmas 4 and 5 (Appendix B). □

Since CIC is more expressive than higher-order predicate logic, adequacy of the representations of meta-proofs does not hold in general; in particular, the ability to eliminate inductive kinds in CIC allows analysis of proof derivations to be used in proof construction, a technique not employed in standard meta-reasoning. This issue does not arise for first-order proof representations like LT (where no constructors have parameters of a function kind), and we do not expect it to be a concern in practice. In cases when it does arise, it could be resolved by using the underlying consistent logic of CIC instead of the meta-logic; for instance in our presentation the question of adequacy is raised because the operational semantics of λ_H is defined in meta-logical terms, but this question would be moot if λ_H and its semantics were defined as CIC terms. To eliminate the interaction with the meta-logic, this approach should be applied all the way down to the hardware specification (as done in some PCC system [2]); we plan to pursue this in the future.

5 CPS Conversion

In this section we show how to perform CPS conversion on λ_H while still preserving proofs represented in the type system. This stage transforms all unconditional control transfers, including function invocation and return, to function calls and gives explicit names to all intermediate computations. The basics of our approach, *i.e.* the target language and the transformation of types, are

shown in this section. The static semantics of the target language and the transformation of terms are given in Appendix D.

We call the target calculus for this phase λ_K, with syntax:

$$
\begin{array}{lll}
(val) & v ::= x \mid \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid \langle X{=}A,\ v{:}A' \rangle \mid \langle v_0,\ \ldots\ v_{n-1} \rangle \\
 & \quad \mid \mathsf{fix}\ x'[X_1{:}A_1,\ \ldots X_n{:}A_n](x{:}A).\, e \\
(exp) & e ::= v[A_1,\ \ldots A_n](v') \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \\
 & \quad \mid \mathsf{let}\ \langle X,\ x \rangle = \mathsf{open}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{sel}[A](v, v')\ \mathsf{in}\ e \\
 & \quad \mid \mathsf{let}\ x = v\ \mathit{aop}\ v'\ \mathsf{in}\ e \mid \mathsf{let}\ x = v\ \mathit{cop}\ v'\ \mathsf{in}\ e \\
 & \quad \mid \mathsf{if}[A, A'](v,\ X_1.\, e_1,\ X_2.\, e_2)
\end{array}
$$

Expressions in λ_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. Multiple arguments may be passed by packing them in a tuple.

λ_K shares the TL type language with λ_H. The types for λ_K all have kind Ω_K which, as in λ_H, is an inductive kind defined in TL. The Ω_K kind has all the constructors of Ω plus one more (func). Since functions in CPS do not return values, the function type constructor of Ω_K has a different kind:

$$\twoheadrightarrow \quad : \quad \Omega_K \to \Omega_K$$

We use the more conventional syntax $A \to \bot$ for $\twoheadrightarrow\ A$. The new constructor func forms the types of function values:

$$\mathsf{func} \quad : \quad \Omega_K \to \Omega_K$$

Every function value is implicitly associated with a closure environment (for all the free variables), so the func constructor is useful in the closure-conversion phase (see Section 6).

Typed CPS conversion involves translating both types and computation terms. Existing algorithms [20, 27] require traversing and transforming every term in the type language (which would include all the proofs in our setting). This is impractical because proofs are large in size, and transforming them can alter their meanings and break the sharing among different intermediate languages.

To see the actual problem, let us convert the λ_H expression $\langle X = A,\ e : B \rangle$ to CPS, assuming that it has type $\exists X : A'.\, B$. We use $\mathcal{K}_{\text{typ}}$ to denote the meta-level translation function for the type language and $\mathcal{K}_{\text{exp}}$ for the computation language. Under existing algorithms, the translation also transforms the witness A:

$$
\begin{array}{l}
\mathcal{K}_{\text{exp}}[\![\, \langle X = A,\ e : B \rangle \,]\!] = \\
\quad \lambda \mathsf{k} : \mathcal{K}_{\text{typ}}[\![\, \exists X : A'.\, B \,]\!]. \\
\qquad \mathcal{K}_{\text{exp}}[\![\, e \,]\!]\ (\lambda x : \mathcal{K}_{\text{typ}}[\![\, [A/X]B \,]\!]. \\
\qquad\qquad \mathsf{k}\ \langle X = \mathcal{K}_{\text{typ}}[\![\, A \,]\!],\ x : \mathcal{K}_{\text{typ}}[\![\, B \,]\!] \rangle)
\end{array}
$$

Here we CPS-convert e and apply it to a continuation, which puts the result of its evaluation in a package and handles it to the return continuation k. With proper definition of $\mathcal{K}_{\text{typ}}$ and assuming that $\mathcal{K}_{\text{typ}}[\![\, X \,]\!] = X$ on all variables X, we can show that the two types $\mathcal{K}_{\text{typ}}[\![\, [A/X]B \,]\!]$ and $[\mathcal{K}_{\text{typ}}[\![\, A \,]\!]/X](\mathcal{K}_{\text{typ}}[\![\, B \,]\!])$ are equivalent (under $=_{\beta\eta\iota}$). Thus the translation preserves typing.

But we do not want to touch the witness A, so the translation function should be defined as follows:

$$
\begin{array}{l}
\mathcal{K}_{\text{exp}}[\![\, \langle X = A,\ e : B \rangle \,]\!] = \\
\quad \lambda \mathsf{k} : \mathcal{K}_{\text{typ}}[\![\, \exists X : A'.\, B \,]\!]. \\
\qquad \mathcal{K}_{\text{exp}}[\![\, e \,]\!]\ (\lambda x : \mathcal{K}_{\text{typ}}[\![\, [A/X]B \,]\!]. \\
\qquad\qquad \mathsf{k}\ \langle X = A,\ x : \mathcal{K}_{\text{typ}}[\![\, B \,]\!] \rangle)
\end{array}
$$

To preserve typing, we have to make sure that the two types $\mathcal{K}_{\text{typ}}[\![\, [A/X]B \,]\!]$ and $[A/X](\mathcal{K}_{\text{typ}}[\![\, B \,]\!])$ are equivalent. This seems impossible to achieve if $\mathcal{K}_{\text{typ}}$ is defined at the meta level.

Our solution is to internalize the definition of $\mathcal{K}_{\text{typ}}$ in our type language. We replace $\mathcal{K}_{\text{typ}}$ by a type function K of kind $\Omega \to \Omega_K$. For readability, we use the pattern-matching syntax, but it can be easily coded using the Elim construct.

$$
\begin{array}{lcl}
\mathsf{K}\ (\mathsf{snat}\ t) & = & \mathsf{snat}\ t \\
\mathsf{K}\ (\mathsf{sbool}\ t) & = & \mathsf{sbool}\ t \\
\mathsf{K}\ (t_1 \to t_2) & = & \mathsf{func}\ ((\mathsf{K}(t_1) \times \mathsf{K_c}(t_2)) \to \bot) \\
\mathsf{K}\ (\mathsf{tup}\ t_1\ t_2) & = & \mathsf{tup}\ t_1\ (\lambda t{:}\mathsf{Nat}.\, \mathsf{K}(t_2\ t)) \\
\mathsf{K}\ (\forall_{\mathsf{Kind}}\ k\ t) & = & \mathsf{func}\ (\forall_{\mathsf{Kind}}\ k\ (\lambda t_1{:}k.\, \mathsf{K_c}(t\ t_1) \to \bot)) \\
\mathsf{K}\ (\forall_{\mathsf{Kscm}}\ z\ t) & = & \mathsf{func}\ (\forall_{\mathsf{Kscm}}\ z\ (\lambda k{:}z.\, \mathsf{K_c}(t\ k) \to \bot)) \\
\mathsf{K}\ (\exists_{\mathsf{Kind}}\ k\ t) & = & \exists_{\mathsf{Kind}}\ k\ (\lambda t_1{:}k.\, \mathsf{K}(t\ t_1)) \\
\mathsf{K}\ (\exists_{\mathsf{Kscm}}\ z\ t) & = & \exists_{\mathsf{Kscm}}\ z\ (\lambda k{:}z.\, \mathsf{K}(t\ k)) \\
 & & \\
\mathsf{K_c} & \equiv & \lambda t{:}\Omega.\, \mathsf{func}\ (\mathsf{K}(t) \to \bot)
\end{array}
$$

The definition of K is in the spirit of the interp function of Crary and Weirich [13]. However interp cannot be used in defining a similar CPS conversion, because its domain does not cover (nor is there an injection to it from) all types appearing in type annotations. In λ_H these types are in the inductive kind Ω and can be analyzed by K. We can now prove $\mathsf{K}\ ([A/X]B) =_{\beta\eta\iota} [A/X](\mathsf{K}\ (B))$ by first reducing B to the normal form B'. Clearly, $\mathsf{K}\ ([A/X]B) =_{\beta\eta\iota} \mathsf{K}\ ([A/X]B')$ and $[A/X](\mathsf{K}\ (B')) =_{\beta\eta\iota} [A/X](\mathsf{K}\ (B))$. We then prove $\mathsf{K}\ ([A/X]B') =_{\beta\eta\iota} [A/X](\mathsf{K}\ (B'))$ by induction over the structure of the normal form B'. The complete CPS-conversion algorithm is given in Appendix D.

6 Closure Conversion

In this section we address the issue of how to make closures explicit for all the CPS terms in λ_K. This stage rewrites all functions so that they contain no free variables. Any variables that appear free in a function value are packaged in an *environment*, which together with the closed code of the function form a *closure*. When a function is applied, the closed code and the environment are extracted from the closure and then the closed code is called with the environment as an additional parameter. Again, the basics of our approach are shown in this section and more details are given in Appendix E.

Our approach to closure conversion is based on Morrisett *et al.* [27], who adopt a type-erasure interpretation of polymorphism. We use the same idea for existential types. The language that we use for this phase is called λ_C with syntax:

$$
\begin{array}{lll}
(val) & v ::= x \mid \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{fix}\ x'[X_1{:}A_1,\ \ldots X_n{:}A_n](x{:}A).\, e \\
 & \quad \mid v[A] \mid \langle v_0,\ \ldots\ v_{n-1} \rangle \mid \langle X{=}A,\ v{:}A' \rangle \\
(exp) & e ::= v\ v' \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{sel}[A](v, v')\ \mathsf{in}\ e \\
 & \quad \mid \mathsf{let}\ \langle X,\ x \rangle = \mathsf{open}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v\ \mathit{aop}\ v'\ \mathsf{in}\ e \\
 & \quad \mid \mathsf{let}\ x = v\ \mathit{cop}\ v'\ \mathsf{in}\ e \mid \mathsf{if}[B, A](v,\ X_1.\, e_1,\ X_2.\, e_2)
\end{array}
$$

λ_C is similar to λ_K, the main difference being that type application and value application are again separate. Type applications are values in λ_C reflecting the fact that they have no runtime effect in a type-erasure interpretation. We use the same kind of types Ω_K as in λ_K. We define the transformation of types as a function $\mathsf{Cl} : \Omega_K \to \Omega_K \to \Omega_K$, the second argument of which represents the type of the environment. As in CPS conversion, we write Cl as a TL function so that the closure-conversion algorithm does not have to traverse proofs represented in the type system.

$$
\begin{array}{lcl}
\mathsf{Cl}\ (\mathsf{snat}\ t) & = & \lambda t' : \Omega_K.\, \mathsf{snat}\ t \\
\mathsf{Cl}\ (\mathsf{sbool}\ t) & = & \lambda t' : \Omega_K.\, \mathsf{sbool}\ t \\
\mathsf{Cl}\ (t \to \bot) & = & \lambda t' : \Omega_K.\, (t' \times \mathsf{Cl}\ (t)\ \bot) \to \bot \\
\mathsf{Cl}\ (\mathsf{func}\ t) & = & \lambda t' : \Omega_K.\, \exists t_1 : \Omega_K.\, (\mathsf{Cl}\ (t)\ t_1 \times t_1) \\
\mathsf{Cl}\ (\mathsf{tup}\ t_1\ t_2) & = & \lambda t' : \Omega_K.\, \mathsf{tup}\ t_1\ (\lambda \mathsf{n} : \mathsf{Nat}.\, \mathsf{Cl}\ (t_2\ \mathsf{n})\ t') \\
\mathsf{Cl}\ (\forall_{\mathsf{Kind}}\ k\ t) & = & \lambda t' : \Omega_K.\, \forall_{\mathsf{Kind}}\ k\ (\lambda t_1 : k.\, \mathsf{Cl}\ (t\ t_1)\ t') \\
\mathsf{Cl}\ (\forall_{\mathsf{Kscm}}\ z\ t) & = & \lambda t' : \Omega_K.\, \forall_{\mathsf{Kind}}\ z\ (\lambda k : z.\, \mathsf{Cl}\ (t\ k)\ t') \\
\mathsf{Cl}\ (\exists_{\mathsf{Kind}}\ k\ t) & = & \lambda t' : \Omega_K.\, \exists_{\mathsf{Kind}}\ k\ (\lambda t_1 : k.\, \mathsf{Cl}\ (t\ t_1)\ t') \\
\mathsf{Cl}\ (\exists_{\mathsf{Kscm}}\ z\ t) & = & \lambda t' : \Omega_K.\, \exists_{\mathsf{Kscm}}\ z\ (\lambda k : z.\, \mathsf{Cl}\ (t\ k)\ t')
\end{array}
$$

7 Related Work

Our type language is a variant of the calculus of constructions [10] extended with inductive definitions (with both small and large elimination) [34, 41]. We omitted parameterized inductive kinds and dependent large elimination to simplify our presentation, however, all our meta-theoretic proofs carry over to a language that includes them. We support η-reduction in our language while the official Coq system does not. The proofs for the properties of TL are adapted from Werner [41] and Geuvers [16]; the main difference is that our language has kind-schema variables and a new product formation rule $(\mathsf{Ext}, \mathsf{Kind})$ which are not in Werner's system.

The Coq proof assistant provides support for extracting programs from proofs [34]. It separates propositions and sets into two distinct universes Prop and Set. We do not distinguish between them because we are not aiming to extract programs from our proofs, instead, we are using proofs as specifications for our computation terms. In fact, the logic in our type language does not have to be constructive; there is no problem with adding classical reasoning to our proof system.

Burstall and McKinna [6] proposed the notion of deliverables, which is essentially the same as our notion of certified binaries. They use dependent strong sum to model each deliverable and give its categorical semantics. Their work does not support programs with effects and has all the problems mentioned in Section 2.3.

Xi and Pfenning's DML [44] is the first language that nicely combines dependent types with programs that may involve effects. Our ideas of using singleton types and lifting the level of the proof language are directly inspired by their work. Xi's system, however, does not support arbitrary propositions and explicit proofs. It also does not define the Ω kind as an inductive definition so it is unclear how it interacts with intensional type analysis [39] and how it preserves proofs during compilation.

We have discussed the relationship between our work and those on PCC, typed assembly languages, and intensional type analysis in Section 1. Inductive definitions subsume and generalize earlier systems on intensional type analysis [21, 13, 39]; the type-analysis construct in the computation language can be eliminated using the technique proposed by Crary *et al.* [15].

Concurrent with our work, Crary and Vanderwaart [11] recently proposed a system called LTT which also aims at adding explicit proofs into typed intermediate languages. LTT uses Linear LF [7] as its proof language. It shares some similarities with our system in that both are using singleton types [44] to circumvent the problems of dependent types. However, since LF does not support the Elim construct on inductive definitions, it is unclear how LTT can support intensional type analysis and type-level primitive recursive functions [14]. In fact, to define Ω as an inductive kind [39], LTT would have to add proof-kind variables and proof-kind polymorphism, which could significantly complicate the meta-theory of its proof language. LTT requires different type languages for different intermediate languages; it is unclear whether it can preserve proofs during CPS and closure conversion. The power of linear reasoning in LTT is desirable for tracking ephemeral properties that hold only for certain program states; we are working on adding such support into our framework.

8 Conclusions

We presented a general framework for explicitly representing propositions and proofs in typed intermediate or assembly languages. We showed how to integrate an entire proof system into our type language and how to perform CPS and closure conversion while still preserving proofs represented in the type system. Our work is a first step toward the goal of building realistic infrastructure for certified programming and certifying compilation.

Our type system is fairly concise and simple with respect to the number of syntactic constructs, yet it is powerful enough to express all the propositions and proofs in the higher-order predicate logic (extended with induction principles). In the future, we would like to use our type system to express advanced program invariants such as those involved in low-level mutable recursive data structures.

Our type language is not designed around any particular programming language. We can use it to typecheck as many different computation languages as we like; all we need is to define the corresponding Ω kind as an inductive definitions. We hope to evolve our framework into a realistic typed common intermediate format.

Acknowledgment

We would like to thank Benjamin Werner for helping us understand the intricacies in the strong normalization proof for the core calculus of inductive constructions.

References

[1] A. W. Appel and E. W. Felten. Models for security policies in proof-carrying code. Technical Report CS-TR-636-01, Princeton Univ., Dept. of Computer Science, March 2001.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 243–253. ACM Press, 2000.

[3] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science (volume 2)*. Oxford Univ. Press, 1991.

[4] H. P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

[5] G. Barthe, J. Hatcliff, and M. Sorensen. CPS translations and applications: the cube and beyond. *Higher Order and Symbolic Computation*, 12(2):125–170, September 1999.

[6] R. Burstall and J. McKinna. Deliverables: an approach to program development in constructions. Technical Report ECS-LFCS-91-133, Univ. of Edinburgh, UK, 1991.

[7] I. Cervesato and F. Pfenning. A linear logical fraemwork. In *Proc. Eleventh IEEE Symposium on Logic in Computer Science*, pages 264–275, July 1996.

[8] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, 2000. ACM Press.

[9] R. Constable. Constructive mathematics as a programming logic I: Some principles of theory. *Ann. of Discrete Mathemathics*, 24, 1985.

[10] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[11] K. Crary and J. Vanderwaart. An expressive, scalable type theory for certified code. Technical Report CMU-CS-01-113, School of Computer Science, Carnegie Mellon Univ., Pittsburg, PA, May 2001.

[12] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.*, pages 262–275. ACM Press, 1999.

[13] K. Crary and S. Weirich. Flexible type analysis. In *Proc. 1999 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 233–248. ACM Press, Sept. 1999.

[14] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 184–198. ACM Press, 2000.

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

[16] H. Geuvers. *Logics and Type Systems*. PhD thesis, Catholic University of Nijmegen, The Netherlands, 1993.

[17] E. Gimenez. A tutorial on recursive types in Coq, May 1998.

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

[19] R. Harper. The practice of type theory. Talk presented at 2000 Alan J. Perlis Symposium, Yale University, New Haven, CT, April 2000.

[20] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Proc. 20th ACM Symp. on Principles of Prog. Lang.*, pages 206–219. ACM Press, 1993.

[21] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd ACM Symp. on Principles of Prog. Lang.*, pages 130–141. ACM Press, 1995.

[22] S. Hayashi. Singleton, union and intersection types for program extraction. In A. R. Meyer, editor, *Proc. International Conference on Theoretical Aspects of Computer Software*, pages 701–730, 1991.

[23] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[24] G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

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

[26] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.*, pages 81–91, New York, 2001. ACM Press.

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

[28] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119, New York, Jan 1997. ACM Press.

[29] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symposium on Operating System Design and Implementation*, pages 229–243, California, 1996. USENIX Association.

[30] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.*, pages 333–344, New York, 1998. ACM Press.

[31] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.

[32] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's type theory*. Oxford University Press, 1990.

[33] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Proc. 16th ACM Symp. on Principles of Prog. Lang.*, pages 89–104, New York, Jan 1989. ACM Press.

[34] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*. LNCS 664, Springer-Verlag, 1993.

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

[36] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 313–323. ACM Press, 1998.

[37] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system of certified binaries. Technical Report YALEU/DCS/TR-1211, Dept. of Computer Science, Yale University, New Haven, CT, March 2001. Available at URL: `flint.cs.yale.edu/flint/publications`.

[38] M. A. Sheldon and D. K. Gifford. Static dependent types for first class modules. In *1990 ACM Conference on Lisp and Functional Programming*, pages 20–29, New York, June 1990. ACM Press.

[39] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Proc. 2000 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 82–93. ACM Press, September 2000.

[40] D. Walker. A type system for expressive security policies. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 254–267. ACM Press, 2000.

[41] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

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

[43] H. Xi and R. Harper. A dependently typed assembly language. In *Proc. 2001 ACM SIGPLAN Int'l Conf. on Functional Prog.*, to appear.

[44] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.*, pages 214–227. ACM Press, 1999.

A Formalization of TL (Details)

In this appendix we supply the rest of the details involved in the formalization of our type language TL. Most of our notations and definitions are directly borrowed from Werner [41]. In addition to the symbols defined in the syntax, we will also use C to denote general terms, Y and Z for variables, and I for inductive definitions.

In order to ensure that the interpretation of inductive definitions remains consistent, and they can be interpreted as terms closed under their introduction rules, we impose *positivity constraints* on the constructors of an inductive definition. The positivity constraints are defined in Definition 2 and 3.

Definition 2 A term A is *strictly positive in* X if A is either X or $\Pi Y : B.\, A'$, where A' is strictly positive in X, X does not occur free in B, and $X \neq Y$.

Definition 3 A term C is a *well-formed constructor kind* for X (written $wfc_X(C)$) if it has one of the following forms:

1. X;
2. $\Pi Y : B.\, C'$, where $Y \neq X$, X is not free in B, and C' is a well-formed constructor kind for X; or
3. $A \to C'$, where A is strictly positive in X and C' is a well-formed constructor kind for X.

Note that in the definition of $wfc_X(C)$, the second clause covers the case where C is of the form $A \to C'$, and X does not occur free in A. Therefore, we only allow the occurrence of X in the non-dependent case.

In the rest of this paper we often write the well-formed constructor kind for X as $\Pi\vec{Y} : \vec{B}.\, X$. We also denote terms that are strictly positive in X by $\Pi\vec{Y} : \vec{B}.\, X$, where X is not free in $\vec{B}$.

Definition 4 Let C be a well-formed constructor kind for X. Then C is of the form $\Pi\vec{Y} : \vec{A}.\, X$. If all the Y's are t's, that is, C is of the form $\Pi\vec{t} : \vec{A}.\, X$, then we say that C is a small constructor kind (or just small constructor when there is no ambiguity) and denote it as $small(C)$.

Our inductive definitions reside in Kind, whereas a small constructor does not make universal quantification over objects of type Kind. Therefore, an inductive definition with small constructors is a predicative definition. While dealing with impredicative inductive definitions, we must forbid projections on universes equal to or bigger than the one inhabited by the definition [17]. In particular, we restrict large elimination to inductive definitions with only small constructors.

Next, we define the set of reductions on our terms. The definition of β- and η-reduction is standard. The ι-reduction defines primitive recursion over inductive objects.

Definition 5 Let C be a well-formed constructor kind for X and let A', B', and I be pseudoterms. We define $\Phi_{X,I,B'}(C, A')$ recursively based on the structure of C:

$$\begin{array}{lcl}
\Phi_{X,I,B'}(X, A') & \stackrel{\text{def}}{=} & A' \\
\Phi_{X,I,B'}(\Pi Y\!:\!B.\, C', A') & \stackrel{\text{def}}{=} & \lambda Y\!:\!B.\, \Phi_{X,I,B'}(C', A'\ Y) \\
\Phi_{X,I,B'}((\Pi \vec{Y}\!:\!\vec{B}.\, X) \to C', A') & \stackrel{\text{def}}{=} & \\
\quad \lambda Z\!:\!(\Pi \vec{Y}\!:\!\vec{B}.\, I).\, \Phi_{X,I,B'}(C', A'\ Z\ (\lambda \vec{Y}\!:\!\vec{B}.\, B'\ (Z\ \vec{Y}))) &&
\end{array}$$

Definition 6 The reduction relations on our terms are defined as:

$$\begin{array}{rcl}
(\lambda X\!:\!A.\, B)\ A' & \rightsquigarrow_\beta & [A'/X]B \\
\lambda X\!:\!A.\, (B\ X) & \rightsquigarrow_\eta & B, \quad \text{if } X \notin FV(B) \\
\mathsf{Elim}[I, A''](\mathsf{Ctor}\,(i, I)\ \vec{A})\{\vec{B}\} & \rightsquigarrow_\iota & (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A} \\
\text{where} \quad I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{C}\} & & \\
\quad B' = \lambda Y\!:\!I.\, (\mathsf{Elim}[I, A''](Y)\{\vec{B}\}) & &
\end{array}$$

By $\rhd_\beta$, $\rhd_\eta$, and $\rhd_\iota$ we denote the relations that correspond to the rewriting of subterms using the relations $\rightsquigarrow_\beta$, $\rightsquigarrow_\eta$, and $\rightsquigarrow_\iota$ respectively. We use $\rightsquigarrow$ and $\rhd$ for the unions of the above relations. We also write $=_{\beta\eta\iota}$ for the reflexive-symmetric-transitive closure of $\rhd$.

Let us examine the ι-reduction in detail. In $\mathsf{Elim}[I, A''](A)\{\vec{B}\}$, the term A of type I is being analyzed. The sequence $\vec{B}$ contains the set of branches for Elim, one for each constructor of I. In the case when $C_i = X$, which implies that A is of the form $\mathsf{Ctor}\,(i, I)$, the Elim just selects the B_i branch:

$$\mathsf{Elim}[I, A''](\mathsf{Ctor}\,(i, I))\{\vec{B}\} \quad \rightsquigarrow_\iota \quad B_i$$

In the case when $C_i = \Pi \vec{Y} : \vec{B}.\, X$ where X does not occur free in $\vec{B}$, then A must be in the form $\mathsf{Ctor}\,(i, I)\ \vec{A}$ with A_i of type B_i. None of the arguments are recursive. Therefore, the Elim should just select the B_i branch and pass the constructor arguments to it. Accordingly, the reduction yields (by expanding the Φ macro):

$$\mathsf{Elim}[I, A''](\mathsf{Ctor}\,(i, I)\ \vec{A})\{\vec{B}\} \quad \rightsquigarrow_\iota \quad B_i\ \vec{A}$$

The recursive case is the most interesting. For simplicity assume that the i-th constructor has the form $\Pi \vec{Y} : \vec{B'}.\, X \to \Pi \vec{Y'} : \vec{B''}.\, X$. Therefore, A is of the form $\mathsf{Ctor}\,(i, I)\ \vec{A}$ with A_1 being the recursive component of type $\Pi \vec{Y} : \vec{B'}.\, X$, and $A_2 \ldots A_n$ being non-recursive. The reduction rule then yields:

$$\begin{array}{l}
\mathsf{Elim}[I, A''](\mathsf{Ctor}\,(i, I)\ \vec{A})\{\vec{B}\} \\
\quad \rightsquigarrow_\iota B_i\ A_1\ (\lambda \vec{Y} : \vec{B'}.\, \mathsf{Elim}[I, A''](A_1\ \vec{Y})\{\vec{B}\})\ A_2 \ldots A_n
\end{array}$$

The Elim construct selects the B_i branch and passes the arguments $A_1, \ldots, A_n$, and the result of recursively processing A_1. In the general case, it would process each recursive argument.

Definition 7 defines the Ψ macro which represents the type of the large Elim branches. Definition 8 defines the ζ macro which represents the type of the small elimination branches. The different cases follow from the ι-reduction rule in Definition 6.

Definition 7 Let C be a well-formed constructor kind for X and let A' and I be two terms. We define $\Psi_{X,I}(C, A')$ recursively based on the structure of C:

$$\begin{array}{lcl}
\Psi_{X,I}(X, A') & \stackrel{\text{def}}{=} & A' \\
\Psi_{X,I}(\Pi Y\!:\!B.\, C', A') & \stackrel{\text{def}}{=} & \Pi Y\!:\!B.\, \Psi_{X,I}(C', A') \\
\Psi_{X,I}(A \to C', A') & \stackrel{\text{def}}{=} & [I/X]A \to [A'/X]A \to \Psi_{X,I}(C', A')
\end{array}$$

where X is not free in B and A is strictly positive in X.

Definition 8 Let C be a well-formed constructor kind for X and let A', I, and B' be terms. We define $\zeta_{X,I}(C, A', B')$ recursively based on the structure of C:

$$\begin{array}{lcl}
\zeta_{X,I}(X, A', B') & \stackrel{\text{def}}{=} & A'\ B' \\
\zeta_{X,I}(\Pi Y\!:\!B.\, C', A', B') & \stackrel{\text{def}}{=} & \Pi Y\!:\!B.\, \zeta_{X,I}(C', A', B'\ Y) \\
\zeta_{X,I}(\Pi \vec{Y}\!:\!\vec{B}.\, X \to C', A', B') & \stackrel{\text{def}}{=} & \\
\quad \Pi Z\!:\!(\Pi \vec{Y}\!:\!\vec{B}.\, I).\, \Pi \vec{Y}\!:\!\vec{B}.\, (A'\ (Z\ \vec{Y})) \to \zeta_{X,I}(C', A', B'\ Z) &&
\end{array}$$

where X is not free in B and $\vec{B}$.

Definition 9 We use $\Delta|_{t,k}$ to denote that the environment does not contain any z variables.

Here are the complete typing rules for TL. The three weakening rules make sure that all variables are bound to the right classes of terms in the context. There are no separate context-formation rules; a context Δ is well-formed if we can derive the judgment $\Delta \vdash \mathsf{Kind} : \mathsf{Kscm}$ (notice we can only add new variables to the context via the weakening rules).

$$\cdot \vdash \mathsf{Kind} : \mathsf{Kscm} \qquad \text{(AX1)}$$

$$\cdot \vdash \mathsf{Kscm} : \mathsf{Ext} \qquad \text{(AX2)}$$

$$\frac{\Delta \vdash C : \mathsf{Kind} \quad \Delta \vdash A : B \quad t \notin Dom(\Delta)}{\Delta, t\!:\!C \vdash A : B} \qquad \text{(WEAK1)}$$

$$\frac{\Delta \vdash C : \mathsf{Kscm} \quad \Delta \vdash A : B \quad k \notin Dom(\Delta)}{\Delta, k\!:\!C \vdash A : B} \qquad \text{(WEAK2)}$$

$$\frac{\Delta \vdash C : \mathsf{Ext} \quad \Delta \vdash A : B \quad z \notin Dom(\Delta)}{\Delta, z\!:\!C \vdash A : B} \qquad \text{(WEAK3)}$$

$$\frac{\Delta \vdash \mathsf{Kind} : \mathsf{Kscm} \qquad X \in Dom(\Delta)}{\Delta \vdash X : \Delta(X)} \qquad \text{(VAR)}$$

$$\frac{\Delta, X\!:\!A \vdash B : B' \qquad \Delta \vdash \Pi X\!:\!A.\, B' : s}{\Delta \vdash \lambda X\!:\!A.\, B : \Pi X\!:\!A.\, B'} \qquad \text{(FUN)}$$

$$\frac{\Delta \vdash A : \Pi X\!:\!B'.\, A' \quad \Delta \vdash B : B'}{\Delta \vdash A\ B : [B/X]A'} \qquad \text{(APP)}$$

$$\frac{\Delta \vdash A : s_1 \quad \Delta, X\!:\!A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X\!:\!A.\, B : s_2} \qquad \text{(PROD)}$$

$$\frac{\textit{for all } i \quad \Delta, X\!:\!\mathsf{Kind} \vdash C_i : \mathsf{Kind} \quad wfc_X(C_i)}{\Delta \vdash \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{C}\} : \mathsf{Kind}} \qquad \text{(IND)}$$

$$\frac{\Delta \vdash I : \mathsf{Kind}\ \textit{where}\ I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{C}\}}{\Delta \vdash \mathsf{Ctor}\,(i, I) : [I/X]C_i} \qquad \text{(CON)}$$

$$\frac{\begin{array}{c}\Delta \vdash A : I \quad \Delta \vdash A' : I \to \mathsf{Kind} \\ \textit{for all } i \quad \Delta \vdash B_i : \zeta_{X,I}(C_i, A', \mathsf{Ctor}\,(i, I))\end{array}}{\begin{array}{c}\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A'\ A \\ \textit{where } I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{C}\}\end{array}} \qquad \text{(ELIM)}$$

$$\frac{\begin{array}{c}\Delta \vdash A : I \quad \Delta|_{t,k} \vdash A' : \mathsf{Kscm} \\ \textit{for all } i \quad \textit{small}(C_i) \quad \Delta \vdash B_i : \Psi_{X,I}(C_i, A')\end{array}}{\begin{array}{c}\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A' \\ \textit{where } I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{C}\}\end{array}} \qquad \text{(L-ELIM)}$$

$$\frac{\Delta \vdash A : B \qquad \Delta \vdash B' : s \qquad \Delta \vdash B : s \qquad B =_{\beta\eta\iota} B'}{\Delta \vdash A : B'} \quad \text{(CONV)}$$

Next we state the formal properties of TL. We omit the proofs due to lack of space and refer the reader to the companion technical report [37] for the details. Our proofs are mostly adapted from Werner [41] and Geuvers [16], but we have to add support for kind-schema variables which is not part of Werner's system.

Theorem 10 (Subject reduction) If the judgment $\Delta \vdash A : B$ is derivable, and if $A \rhd A'$ and $\Delta \rhd \Delta'$, then the following are derivable: $\Delta \vdash A' : B$ and $\Delta' \vdash A : B$.

Theorem 11 (Strong normalization) All well typed terms are strongly normalizing.

Theorem 12 (Church-Rosser) Let $\Delta \vdash A : B$ and $\Delta \vdash A' : B$ be two derivable judgments. If $A =_{\beta\eta\iota} A'$, and if A and A' are in normal form, then $A = A'$.

Theorem 13 (Consistency of the logic) There exists no term A for which $\cdot \vdash A : \mathsf{False}$.

B Properties of λ_H

The proof of the following lemma is by induction on the structure of typing derivations.

Lemma 2 If $\Delta, X : B; \Gamma \vdash e : A'$ and $\Delta \vdash A : B$, then $\Delta; \Gamma \vdash [A/X]e : [A/X]A'$.

We also need a proposition guaranteeing that equivalence of constructor applications implies equivalence of their arguments; it is a corollary of the confluence of TL (Theorem 12).

Lemma 3 If $\mathsf{Ctor}\,(i, I)\ \vec{A} =_{\beta\eta\iota} \mathsf{Ctor}\,(i', I')\ \vec{A'}$, then $i = i'$ and $I =_{\beta\eta\iota} I'$ and $\vec{A} =_{\beta\eta\iota} \vec{A'}$.

Lemma 4 (Progress) If $\cdot;\cdot \vdash e : A$, then either e is a value, or there exists e' such that $e \mapsto e'$.

Proof sketch By standard techniques [42] using induction on computation terms. Due to the transitivity of $=_{\beta\eta\iota}$ any derivation of $\Delta; \Gamma \vdash e : A$ can be converted to a standard form in which there is an application of rule E-CONV at its root, whose first premise ends with an instance of a rule other than E-CONV, all of whose term derivation premises are in standard form.

We omit the proofs for the cases of standard constructs and the induction on the structure of evaluation contexts. The interesting case is that of the dependently typed sel.

If $e = \mathsf{sel}[A'](v, v')$, by inspection of the typing rules the derivation of $\cdot;\cdot \vdash e : A$ in standard form must have an instance of rule E-SEL in the premise of its root. Hence the subderivation for v must assign to it a tuple type, and the whole derivation has the form

$$\frac{\dfrac{\dfrac{\mathcal{D}}{\cdot;\cdot \vdash v : \mathsf{tup}\ A_2\ A''} \quad \dfrac{\mathcal{D}'}{\cdot;\cdot \vdash v' : \mathsf{snat}\ A_1} \quad \dfrac{\mathcal{E}}{\cdot \vdash A' : \mathsf{LT}\ A_1\ A_2}}{\cdot;\cdot \vdash \mathsf{sel}[A'](v, v') : A''\ A_1}}{\cdot;\cdot \vdash \mathsf{sel}[A'](v, v') : A}$$

where $A =_{\beta\eta\iota} A''\ A_1$. By inspection of the typing rules, rules other than E-CONV assign to all values types which are applications of constructors of Ω. Since the derivation $\mathcal{D}$ is in standard form, it ends with an E-CONV, in the premise of which another rule assigns v a type $\beta\eta\iota$-equivalent to $\mathsf{tup}\ A_2\ A''$. Then by Lemma 3 this type must be an application of tup, and again by inspection the only rule which applies is E-TUP, which implies $v = \langle v_0, \ldots v_{n-1}\rangle$, and the derivation $\mathcal{D}$ must have the form

$$\frac{\forall i < n \ \dfrac{\mathcal{D}_i}{\cdot;\cdot \vdash v_i : A_1''\ \widehat{i}}}{\cdot;\cdot \vdash \langle v_0, \ldots v_{n-1}\rangle : \mathsf{tup}\ \widehat{n}\ A_1''}$$

Also by Lemma 3 $A_2 =_{\beta\eta\iota} \widehat{n}$. Similarly the only rule assigning to a value a type convertible to that in the conclusion of $\mathcal{D}'$ is E-NAT, hence $A_1 =_{\beta\eta\iota} \widehat{m}$ for some $m \in \mathbb{N}$, and $v' = \overline{m}$. Then, by adequacy of LT (Lemma 1(3)), the conclusion of $\mathcal{E}$ implies that $m < n$. Hence by rule R-SEL $e \mapsto v_m$. □

Lemma 5 (Subject Reduction) If $\cdot;\cdot \vdash e : A$ and $e \mapsto e'$, then $\cdot;\cdot \vdash e' : A$.

Proof sketch Since evaluation contexts bind no variables, it suffices to prove subject reduction for $\hookrightarrow$ and a standard term substitution lemma. We show only some cases of redexes involving sel and if.

- The derivation for $e = \mathsf{sel}[A'](\langle v_0, \ldots v_{n-1}\rangle, \overline{m})$ in standard form has the shape

$$\frac{\dfrac{\dfrac{\dfrac{\forall i < n \ \dfrac{\mathcal{D}_i}{\cdot;\cdot \vdash v_i : A_1''\ \widehat{i}}}{\cdot;\cdot \vdash \langle \vec{v}\rangle : \mathsf{tup}\ \widehat{n}\ A_1''} \quad \dfrac{\dfrac{\mathcal{D}'}{\cdot;\cdot \vdash \overline{m} : \mathsf{snat}\ \widehat{m}}}{\cdot;\cdot \vdash \overline{m} : \mathsf{snat}\ A_1} \quad \dfrac{\mathcal{E}}{\cdot \vdash A' : \mathsf{LT}\ A_1\ A_2}}{\cdot;\cdot \vdash \mathsf{sel}[A'](\langle v_0, \ldots v_{n-1}\rangle, \overline{m}) : A''\ A_1}}{\cdot;\cdot \vdash \mathsf{sel}[A'](\langle v_0, \ldots v_{n-1}\rangle, \overline{m}) : A}$$

  where $A =_{\beta\eta\iota} A''\ A_1$, $A_1'' =_{\beta\eta\iota} A''$, and $A_1 =_{\beta\eta\iota} \widehat{m}$. Since $e \mapsto e'$ only by rule R-SEL, we have $m < n$ and $e' = v_m$, so from $\mathcal{D}_m$ and $A_1''\ \widehat{m} =_{\beta\eta\iota} A''\ \widehat{m} =_{\beta\eta\iota} A''\ A_1 =_{\beta\eta\iota} A$ we obtain a derivation of $\cdot;\cdot \vdash e' : A$.

- In the case of if the standard derivation $\mathcal{D}$ of

$$\cdot;\cdot \vdash \mathsf{if}[B, A'](\mathsf{tt},\ X_1.\, e_1,\ X_2.\, e_2) : A$$

  ends with an instance of E-CONV, preceded by an instance of E-IF. Using the notation from Figure 5, from the premises of this rule it follows that we have a derivation $\mathcal{E}$ of $\cdot \vdash A' : B\ A''$, and $A'' =_{\beta\eta\iota} \mathsf{true}$ (since rule E-TRUE assigns sbool true to tt), hence we have $\cdot \vdash A' : B\ \mathsf{true}$ by CONV. By Lemma 2 from $\mathcal{E}$ and the derivation of $X_1 : B\ \mathsf{true}; \cdot \vdash e_1 : A$ (provided as another premise), since X_1 is not free in A (ensured by the premise $\cdot \vdash A : \Omega$) we obtain a derivation of $\cdot;\cdot \vdash [A'/X_1]e_1 : A$. □

C Example of Proof Construction

Here we show the type term ltPrf which generates the proof of the proposition $\mathsf{LTOrTrue}\ t'\ t\ (\mathsf{lt}\ t'\ t)$, needed in the sumVec example in Section 4. We first present a Church encoding of the kind term LT and its "constructors" ltzs and ltss.

$$\begin{aligned}
&\mathsf{LT} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}\\
&\mathsf{LT} = \lambda t : \mathsf{Nat}.\ \lambda t' : \mathsf{Nat}.\\
&\qquad \Pi R : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}.\\
&\qquad (\Pi t : \mathsf{Nat}.\ R\ \mathsf{zero}\ (\mathsf{succ}\ t)) \to\\
&\qquad (\Pi t : \mathsf{Nat}.\ \Pi t' : \mathsf{Nat}.\ R\ t\ t' \to R\ (\mathsf{succ}\ t)\ (\mathsf{succ}\ t')) \to\\
&\qquad R\ t\ t'
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{ltzs} : \Pi t{:}\mathsf{Nat}.\ \mathsf{LT}\ \mathsf{zero}\ (\mathsf{succ}\ t)\\
&\mathsf{ltzs} = \lambda t{:}\mathsf{Nat}.\ \lambda R{:}\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Kind}.\\
&\qquad \lambda z{:}(\Pi t{:}\mathsf{Nat}.\ R\ \mathsf{zero}\ (\mathsf{succ}\ t)).\\
&\qquad \lambda s{:}(\Pi t{:}\mathsf{Nat}.\ \Pi t'{:}\mathsf{Nat}.\ R\ t\ t'\to R\ (\mathsf{succ}\ t)\ (\mathsf{succ}\ t')).\\
&\qquad z\ t\\
&\mathsf{ltss} : \Pi t{:}\mathsf{Nat}.\ \Pi t'{:}\mathsf{Nat}.\ \mathsf{LT}\ t\ t'\to\mathsf{LT}\ (\mathsf{succ}\ t)\ (\mathsf{succ}\ t')\\
&\mathsf{ltss} = \lambda t{:}\mathsf{Nat}.\ \lambda t'{:}\mathsf{Nat}.\ \lambda p{:}\mathsf{LT}\ t\ t'.\ \lambda R{:}\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Kind}.\\
&\qquad \lambda z{:}(\Pi t{:}\mathsf{Nat}.\ R\ \mathsf{zero}\ (\mathsf{succ}\ t)).\\
&\qquad \lambda s{:}(\Pi t{:}\mathsf{Nat}.\ \Pi t'{:}\mathsf{Nat}.\ R\ t\ t'\to R\ (\mathsf{succ}\ t)\ (\mathsf{succ}\ t')).\\
&\qquad s\ t\ t'\ (p\ R\ z\ s)
\end{aligned}
$$

Next we define dependent conditionals on kinds Nat and Bool.

$$
\begin{aligned}
&\mathsf{dep_ifez} : \Pi t{:}\mathsf{Nat}.\ \Pi k{:}\mathsf{Nat}\to\mathsf{Kind}.\\
&\qquad\qquad k\ \mathsf{zero}\to(\Pi t'{:}\mathsf{Nat}.\ k\ (\mathsf{succ}\ t'))\to k\ t\\
&\mathsf{dep_ifez\ zero} \quad = \lambda k{:}\mathsf{Nat}\to\mathsf{Kind}.\ \lambda t_1{:}k\ \mathsf{zero}.\\
&\qquad\qquad \lambda t_2{:}(\Pi t'{:}\mathsf{Nat}.\ k\ (\mathsf{succ}\ t')).\ t_1\\
&\mathsf{dep_ifez}\ (\mathsf{succ}\ t) = \lambda k{:}\mathsf{Nat}\to\mathsf{Kind}.\ \lambda t_1{:}k\ \mathsf{zero}.\\
&\qquad\qquad \lambda t_2{:}(\Pi t'{:}\mathsf{Nat}.\ k\ (\mathsf{succ}\ t')).\ t_2\ t
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{dep_if} : \Pi t{:}\mathsf{Bool}.\ \Pi k{:}\mathsf{Bool}\to\mathsf{Kind}.\ k\ \mathsf{true}\to k\ \mathsf{false}\to k\ t\\
&\mathsf{dep_if\ true} = \lambda k{:}\mathsf{Bool}\to\mathsf{Kind}.\ \lambda t_1{:}k\ \mathsf{true}.\ \lambda t_2{:}k\ \mathsf{false}.\ t_1\\
&\mathsf{dep_if\ false} = \lambda k{:}\mathsf{Bool}\to\mathsf{Kind}.\ \lambda t_1{:}k\ \mathsf{true}.\ \lambda t_2{:}k\ \mathsf{false}.\ t_2
\end{aligned}
$$

Finally, some abbreviations, and then the proof generator itself.

$$
\begin{aligned}
&\mathsf{LTcond} : \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Kind}\\
&\mathsf{LTcond} = \lambda t'{:}\mathsf{Nat}.\ \lambda t{:}\mathsf{Nat}.\ \mathsf{LTOrTrue}\ t'\ t\ (\mathsf{lt}\ t'\ t)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{LTimp} : \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Bool}\to\mathsf{Kind}\\
&\mathsf{LTimp} = \lambda t'{:}\mathsf{Nat}.\ \lambda t{:}\mathsf{Nat}.\ \lambda t''{:}\mathsf{Bool}.\\
&\qquad \mathsf{LTOrTrue}\ t'\ t\ t''\to\mathsf{LTOrTrue}\ (\mathsf{succ}\ t')\ (\mathsf{succ}\ t)\ t''
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ltPrf} : \Pi t'{:}\mathsf{Nat}.\ \Pi t{:}\mathsf{Nat}.\ \mathsf{LTcond}\ t'\ t\\
&\mathsf{ltPrf} = \lambda t'{:}\mathsf{Nat}.\ \lambda t{:}\mathsf{Nat}.\\
&\qquad \mathsf{Elim}[\mathsf{Nat}, \lambda t_1'{:}\mathsf{Nat}.\ \Pi t_1{:}\mathsf{Nat}.\ \mathsf{LTcond}\ t_1'\ t_1](t')\{\\
&\qquad\quad \lambda t_1{:}\mathsf{Nat}.\ \mathsf{dep_ifez}\ t_1\ (\mathsf{LTcond\ zero})\ \mathsf{id}\ \mathsf{ltzs};\\
&\qquad\quad \lambda t_1'{:}\mathsf{Nat}.\ \lambda t_P{:}(\Pi t_1{:}\mathsf{Nat}.\ \mathsf{LTcond}\ t_1'\ t_1).\ \lambda t_1{:}\mathsf{Nat}.\\
&\qquad\qquad \mathsf{dep_ifez}\ t_1\\
&\qquad\qquad\qquad (\mathsf{LTcond}\ (\mathsf{succ}\ t_1'))\\
&\qquad\qquad\qquad \mathsf{id}\\
&\qquad\qquad\qquad (\lambda t_1{:}\mathsf{Nat}.\ \mathsf{dep_if}\ (\mathsf{lt}\ t_1'\ t_1)\\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{LTimp}\ t_1'\ t_1)\\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{ltss}\ t_1'\ t_1)\\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{id\ True})\\
&\qquad\qquad\qquad\qquad\qquad (t_P\ t_1))\}
\end{aligned}
$$

D CPS Conversion (Details)

We start by defining a version of λ_H 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 syntactic instead of typing derivations.

$$
\begin{aligned}
(\textit{exp}) \quad & e ::= \bar{e}^A\\
& \bar{e} ::= x \mid \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid f \mid \mathsf{fix}\ x{:}A.\ f \mid e\ e' \mid e[A]\\
& \qquad \mid \langle X{=}A,\ e{:}A'\rangle \mid \mathsf{open}\ e\ \mathsf{as}\ \langle X,\ x\rangle\ \mathsf{in}\ e'\\
& \qquad \mid \langle e_0,\ \ldots\ e_{n-1}\rangle \mid \mathsf{sel}[A](e, e') \mid e\ \mathit{aop}\ e'\\
& \qquad \mid e\ \mathit{cop}\ e' \mid \mathsf{if}[A, A'](e,\ X_1.\ e_1,\ X_2.\ e_2)\\
(\textit{fun}) \quad & f ::= \bar{f}^A\\
& \bar{f} ::= \lambda x{:}A.\ e \mid \Lambda X{:}A.\ f
\end{aligned}
$$

The target language λ_K of the CPS conversion stage has been defined in Section 5. We use the following syntactic sugar to denote non-recursive function definitions and value applications in λ_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 λ_K we use two forms of judgments. As in λ_H, the judgment $\Delta;\ \Gamma\ \vdash_K\ v : A$ indicates that the value v is well formed and of type A in the type and value contexts Δ and Γ respectively. Moreover, $\Delta;\ \Gamma\ \vdash_K\ e$ indicates that the expression e is well formed in Δ and Γ. In both forms of judgments, we omit the subscript from $\vdash_K$ when it can be deduced from the context.

The static semantics of λ_K is specified by the following formation rules (we omit the rules for environment formation, variables, constants, tuples, packages, and type conversion on values, which are the same as in λ_H):

$$
\frac{\begin{array}{c}\textit{for all } i \in \{1\ldots n\} \qquad \Delta\ \vdash\ A_i : s_i\\ \Delta, X_1{:}A_1\ldots, X_n{:}A_n\ \vdash\ A : \Omega\\ \Delta, X_1{:}A_1\ldots, X_n{:}A_n;\ \Gamma, x'{:}A', x{:}A\ \vdash\ e\end{array}}{\begin{array}{l}\Delta;\ \Gamma\ \vdash\ \mathsf{fix}\ x'[X_1{:}A_1,\ \ldots X_n{:}A_n](x{:}A).\ e : A'\\ \textit{where}\\ \quad A' = \mathsf{func}\ (\forall_{s_1} X_1{:}A_1.\ldots\forall_{s_n} X_n{:}A_n.\ A\to\bot)\end{array}} \qquad \text{(K-FIX)}
$$

$$
\frac{\begin{array}{c}\textit{for all } i \in \{1\ldots n\} \qquad \Delta\ \vdash\ A_i : B_i\\ \Delta;\ \Gamma\vdash v' : \mathsf{func}\,(\forall_{s_1} X_1{:}B_1.\ldots\forall_{s_n} X_n{:}B_n.\ A\to\bot)\\ \Delta;\ \Gamma\ \vdash\ v : [A_1/X_1]\ldots[A_n/X_n]A\end{array}}{\Delta;\ \Gamma\ \vdash\ v'[A_1,\ \ldots A_n](v)} \qquad \text{(K-APP)}
$$

$$
\frac{\Delta;\ \Gamma\ \vdash\ v : A \qquad \Delta;\ \Gamma, x{:}A\ \vdash\ e}{\Delta;\ \Gamma\ \vdash\ \mathsf{let}\ x = v\ \mathsf{in}\ e} \qquad \text{(K-VAL)}
$$

$$
\frac{\begin{array}{c}\Delta;\ \Gamma\ \vdash\ v : \mathsf{tup}\ A''\ B \qquad \Delta;\ \Gamma\ \vdash\ v' : \mathsf{snat}\ A'\\ \Delta\ \vdash\ A : \mathsf{LT}\ A'\ A'' \qquad \Delta;\ \Gamma, x{:}B\ A'\ \vdash\ e\end{array}}{\Delta;\ \Gamma\ \vdash\ \mathsf{let}\ x = \mathsf{sel}[A](v, v')\ \mathsf{in}\ e} \qquad \text{(K-SEL)}
$$

$$
\frac{\begin{array}{c}\Delta;\ \Gamma\ \vdash\ v : \exists_s Y{:}B.\ A\\ \Delta, X{:}B;\ \Gamma, x{:}[X/Y]A\ \vdash\ e\end{array}}{\Delta;\ \Gamma\ \vdash\ \mathsf{let}\ \langle X,\ x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e}\ \begin{pmatrix} X \notin \Delta \\ s \neq \mathsf{Ext}\end{pmatrix} \qquad \text{(K-OPEN)}
$$

$$
\frac{\begin{array}{c}\Delta;\ \Gamma\ \vdash\ v : \mathsf{snat}\ A \qquad \Delta;\ \Gamma\ \vdash\ v' : \mathsf{snat}\ A'\\ \Delta;\ \Gamma, x{:}\mathsf{snat}\ (\mathsf{plus}\ A\ A')\ \vdash\ e\end{array}}{\Delta;\ \Gamma\ \vdash\ \mathsf{let}\ x = v + v'\ \mathsf{in}\ e} \qquad \text{(K-ADD)}
$$

$$
\frac{\begin{array}{c}\Delta;\ \Gamma\ \vdash\ v : \mathsf{snat}\ A \qquad \Delta;\ \Gamma\ \vdash\ v' : \mathsf{snat}\ A'\\ \Delta;\ \Gamma, x{:}\mathsf{sbool}\ (\mathsf{lt}\ A\ A')\ \vdash\ e\end{array}}{\Delta;\ \Gamma\ \vdash\ \mathsf{let}\ x = v < v'\ \mathsf{in}\ e} \qquad \text{(K-LT)}
$$

$$
\frac{\begin{array}{c}\Delta\ \vdash\ B : \mathsf{Bool}\to\mathsf{Kind} \qquad \Delta\ \vdash\ A : B\ A'\\ \Delta;\ \Gamma\ \vdash\ v : \mathsf{sbool}\ A'\\ \Delta, X_1{:}B\ \mathsf{true};\ \Gamma\ \vdash\ e_1 \qquad \Delta, X_2{:}B\ \mathsf{false};\ \Gamma\ \vdash\ e_2\end{array}}{\Delta;\ \Gamma\ \vdash\ \mathsf{if}[B, A](v,\ X_1.\ e_1,\ X_2.\ e_2)} \qquad \text{(K-IF)}
$$

Except for the rules K-FIX and K-APP, which must take into account the presence of func, the static semantics for λ_K is a natural consequence of the static semantics for λ_H.

The definition of the CPS transformation for computation terms of λ_H to computation terms of λ_K is given in Figure 6, where we use the abbreviations introduced in Section 5.

Proposition 14 (Type Correctness of CPS Conversion)
If $\cdot;\cdot\vdash_H\ e : A$, then $\cdot;\cdot\vdash_K\ \mathcal{K}_{\mathrm{exp}}[\![\bar{e}^A]\!] : \mathsf{func}\ (\mathsf{K_c}(A)\to\bot)$.

$$
\begin{aligned}
\mathcal{K}_{\text{fval}}[\![(\lambda x\!:\!A.\, e^B)^{A\to B}]\!] &= \lambda x_{\text{arg}}\!:\!\mathsf{K}(A)\times\mathsf{K}_{\text{c}}(B).\\
&\quad \mathsf{let}\ x = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{0}\ \widehat{2}](x_{\text{arg}}, \overline{0})\ \mathsf{in}\\
&\quad \mathsf{let}\ \mathsf{k} = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{1}\ \widehat{2}](x_{\text{arg}}, \overline{1})\ \mathsf{in}\\
&\quad \mathcal{K}_{\text{exp}}[\![e^B]\!]\ \mathsf{k}\\
\mathcal{K}_{\text{fval}}[\![(\Lambda X\!:\!A.\, f^B)^{\forall_s X A.\, B}]\!] &=\\
&\quad \Lambda X\!:\!A.\, \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(B).\, \mathsf{k}\ (\mathcal{K}_{\text{fval}}[\![f^B]\!])\\
\mathcal{K}_{\text{exp}}[\![\bar{e}^A]\!] &= \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A).\, \mathsf{k}\ (\bar{e})\\
&\quad \text{for } \bar{e}^A \text{ one of } x^A,\ \overline{n}^{\mathsf{snat}\ \widehat{n}},\ \mathsf{tt}^{\mathsf{sbool\ true}},\ \mathsf{ff}^{\mathsf{sbool\ false}}\\
\mathcal{K}_{\text{exp}}[\![f^A]\!] &= \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A).\, \mathsf{k}\ (\mathcal{K}_{\text{fval}}[\![f^A)]\!]\\
\mathcal{K}_{\text{exp}}[\![(\mathsf{fix}\ x\!:\!A.\, f^A)^A]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A).\, \mathsf{k}\ (\mathsf{fix}\ x[](\mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A)).\, \mathsf{k}\ (\mathcal{K}_{\text{fval}}[\![f^A]\!]))\\
\mathcal{K}_{\text{exp}}[\![({e_1}^{A\to B}\ {e_2}^A)^B]\!] &= \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(B).\\
&\quad \mathcal{K}_{\text{exp}}[\![{e_1}^{A\to B}]\!]\ (\lambda x_1\!:\!\mathsf{K}(A\to B).\\
&\quad \mathcal{K}_{\text{exp}}[\![{e_2}^A]\!]\ (\lambda x_2\!:\!\mathsf{K}(A).\\
&\quad x_1\ \langle x_2, \mathsf{k}\rangle))\\
\mathcal{K}_{\text{exp}}[\![(e^{\forall_s\ A'\ B}[A])^{B\ A}]\!] &= \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(B\ A).\\
&\quad \mathcal{K}_{\text{exp}}[\![e^{\forall_s\ A'\ B}]\!]\ (\lambda x\!:\!\mathsf{K}(\forall_s\ A'\ B).\\
&\quad x[A](\mathsf{k}))\\
\mathcal{K}_{\text{exp}}[\![\langle e_0^{A_0},\ \ldots\ e_{n-1}^{A_{n-1}}\rangle^A]\!] &= \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A).\\
&\quad \mathcal{K}_{\text{exp}}[\![e_0^{A_0}]\!]\ (\lambda x_0\!:\!\mathsf{K}(A_0).\\
&\quad \vdots\\
&\quad \mathcal{K}_{\text{exp}}[\![e_{n-1}^{A_{n-1}}]\!]\ (\lambda x_{n-1}\!:\!\mathsf{K}(A_{n-1}).\\
&\quad \mathsf{k}\ \langle x_0,\ \ldots x_{n-1}\rangle)\ldots)\\
\mathcal{K}_{\text{exp}}[\![\mathsf{sel}[A]({e_1}^{\mathsf{tup}\ A''\ B}, {e_2}^{\mathsf{snat}\ A'})^{B\ A'}]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(B\ A').\, \mathcal{K}_{\text{exp}}[\![{e_1}^{\mathsf{tup}\ A''\ B}]\!]\ (\lambda x_1\!:\!\mathsf{K}(\mathsf{tup}\ A''\ B).\\
&\quad \mathcal{K}_{\text{exp}}[\![{e_2}^{\mathsf{snat}\ A'}]\!]\ (\lambda x_2\!:\!\mathsf{K}(\mathsf{snat}\ A').\\
&\quad \mathsf{let}\ x' = \mathsf{sel}[A](x_1, x_2)\ \mathsf{in}\ \mathsf{k}\ x'))\\
\mathcal{K}_{\text{exp}}[\![\langle X = A,\ e^{[A/X]B}\!:\!B\rangle^{A'}]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A').\, \mathcal{K}_{\text{exp}}[\![e^{[A/X]B}]\!]\ (\lambda x\!:\!\mathsf{K}([A/X]B).\\
&\quad \mathsf{k}\ \langle X = A,\ x\!:\!\mathsf{K}(B)\rangle)\\
\mathcal{K}_{\text{exp}}[\![(\mathsf{open}\ {e_1}^{\exists_s Y\!:\!A'.\, B}\ \mathsf{as}\ \langle X,\ x\rangle\ \mathsf{in}\ {e_2}^A)^A]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A).\, \mathcal{K}_{\text{exp}}[\![{e_1}^{\exists_s Y\!:\!A'.\, B}]\!]\ (\lambda x_1\!:\!\mathsf{K}(\exists_s Y\!:\!A'.\, B).\\
&\quad \mathsf{let}\ \langle X,\ x\rangle = \mathsf{open}\ x_1\ \mathsf{in}\ \mathcal{K}_{\text{exp}}[\![{e_2}^A]\!]\ \mathsf{k})\\
\mathcal{K}_{\text{exp}}[\![({e_1}^{\mathsf{snat}\ A} + {e_2}^{\mathsf{snat}\ A'})^{\mathsf{snat}\ (\mathsf{plus}\ A\ A')}]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(\mathsf{snat}\ (\mathsf{plus}\ A\ A')).\, \mathcal{K}_{\text{exp}}[\![{e_1}^{\mathsf{snat}\ A}]\!]\ (\lambda x_1\!:\!\mathsf{K}(\mathsf{snat}\ A).\\
&\quad \mathcal{K}_{\text{exp}}[\![{e_2}^{\mathsf{snat}\ A'}]\!]\ (\lambda x_2\!:\!\mathsf{K}(\mathsf{snat}\ A').\\
&\quad \mathsf{let}\ x' = x_1 + x_2\ \mathsf{in}\ \mathsf{k}\ x'))\\
\mathcal{K}_{\text{exp}}[\![({e_1}^{\mathsf{snat}\ A} < {e_2}^{\mathsf{snat}\ A'})^{\mathsf{sbool}\ (\mathsf{lt}\ A\ A')}]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(\mathsf{sbool}\ (\mathsf{lt}\ A\ A')).\, \mathcal{K}_{\text{exp}}[\![{e_1}^{\mathsf{snat}\ A}]\!]\ (\lambda x_1\!:\!\mathsf{K}(\mathsf{snat}\ A).\\
&\quad \mathcal{K}_{\text{exp}}[\![{e_2}^{\mathsf{snat}\ A'}]\!]\ (\lambda x_2\!:\!\mathsf{K}(\mathsf{snat}\ A').\\
&\quad \mathsf{let}\ x' = x_1 < x_2\ \mathsf{in}\ \mathsf{k}\ x'))\\
\mathcal{K}_{\text{exp}}[\![(\mathsf{if}[B, A](e^{\mathsf{sbool}\ A''},\ X_1.\, {e_1}^{A'},\ X_2.\, {e_2}^{A'}))^{A'}]\!] &=\\
&\quad \lambda \mathsf{k}\!:\!\mathsf{K}_{\text{c}}(A').\, \mathcal{K}_{\text{exp}}[\![e^{\mathsf{sbool}\ A''}]\!]\ (\lambda x\!:\!\mathsf{K}(\mathsf{sbool}\ A'').\\
&\quad \mathsf{if}[B, A](x,\ X_1.\, \mathcal{K}_{\text{exp}}[\![{e_1}^{A'}]\!]\ \mathsf{k},\ X_2.\, \mathcal{K}_{\text{exp}}[\![{e_2}^{A'}]\!]\ \mathsf{k}))
\end{aligned}
$$

Figure 6: CPS conversion: from λ_H to λ_K.

E Closure Conversion (Details)

The main difference in the static semantics between λ_K and λ_C is that in the latter the body of a function must not contain free type or term variables. This is formalized in the rule C-FIX below. The rules C-TAPP and C-APP corresponding to the separate type and value application in λ_C are standard.

$$
\frac{\begin{array}{c}\textit{for all } i < n \qquad \cdot \vdash A_i : s_i\\ \cdot, X_1\!:\!A_1 \ldots, X_n\!:\!A_n \vdash A : \Omega\\ \cdot, X_1\!:\!A_1 \ldots, X_n\!:\!A_n;\ \cdot, x'\!:\!B, x\!:\!A \vdash e\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash \mathsf{fix}\ x'[X_1\!:\!A_1,\ \ldots X_n\!:\!A_n](x\!:\!A).\, e : B\\ \textit{where } B = \forall_{s_1} X_1\!:\!A_1 \ldots \forall_{s_n} X_n\!:\!A_n.\, A\to\bot\end{array}} \quad \text{(C-FIX)}
$$

$$
\frac{\Delta;\Gamma \vdash v : \forall_s X\!:\!A'.\, B \qquad \Delta \vdash A : A'}{\Delta;\Gamma \vdash v[A] : [A/X]B} \quad \text{(C-TAPP)}
$$

$$
\frac{\Delta;\Gamma \vdash v_1 : A\to\bot \qquad \Delta;\Gamma \vdash v_2 : A}{\Delta;\Gamma \vdash v_1\ v_2} \quad \text{(C-APP)}
$$

The definition of the closure transformation for the computation terms of λ_K is given in Figure 7.

$$
\begin{aligned}
\mathcal{C}_{\text{val}}[\![v]\!] &= v, \qquad \text{for } v \text{ one of } x,\ \overline{n},\ \mathsf{tt},\ \mathsf{ff}\\
\mathcal{C}_{\text{val}}[\![\langle v_0,\ \ldots\ v_{n-1}\rangle]\!] &= \langle \mathcal{C}_{\text{val}}[\![v_0]\!],\ \ldots \mathcal{C}_{\text{val}}[\![v_{n-1}]\!]\rangle\\
\mathcal{C}_{\text{val}}[\![\langle X = A,\ v\!:\!B\rangle]\!] &= \langle X = A,\ \mathcal{C}_{\text{val}}[\![v]\!]\!:\!\mathsf{Cl}\ (B)\ \bot\rangle\\
\mathcal{C}_{\text{val}}[\![\mathsf{fix}\ x'[X_1\!:\!A_1,\ \ldots X_n\!:\!A_n](x\!:\!A).\, e]\!] &=\\
&\quad \langle X = A_{\text{env}},\ \langle v_{\text{code}}[Y_1]\ldots[Y_m], v_{\text{env}}\rangle\!:\!A_X\rangle\\
&\quad \text{where}\\
&\quad A_X = A'_X \times X\\
&\quad A'_X = \forall_{s_1} X_1\!:\!A_1 \ldots \forall_{s_n} X_n\!:\!A_n.\, (X \times \mathsf{Cl}\ (A)\ \bot)\to\bot\\
&\quad \{x_0^{A'_0},\ \ldots x_{k-1}^{A'_{k-1}}\} = FV(e) - \{x,\ x'\}\\
&\quad \{Y_1^{B'_1},\ \ldots Y_m^{B'_m}\} =\\
&\qquad FTV(\mathsf{fix}\ x'[X_1\!:\!A_1,\ \ldots X_n\!:\!A_n](x\!:\!A).\, e)\\
&\quad A_{\text{env}} = \mathsf{Cl}\ (\mathsf{tup}\ \widehat{k}\ (\mathsf{nth}\ (A'_0{::}\ldots A'_{k-1}{::}\mathsf{nil})))\ \bot\\
&\quad v_{\text{env}} = \langle x_0\ \ldots x_{k-1}\rangle\\
&\quad v_{\text{code}} = \mathsf{fix}\ v_{\text{fix}}[Y_1\!:\!B'_1,\ \ldots Y_m\!:\!B'_m, X_1\!:\!A_1,\ \ldots X_n\!:\!A_n]\\
&\qquad (x_{\text{arg}}\!:\!A_{\text{env}} \times \mathsf{Cl}\ (A)\ \bot).\\
&\qquad \mathsf{let}\ x_{\text{env}} = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{0}\ \widehat{2}](x_{\text{arg}}, \overline{0})\ \mathsf{in}\\
&\qquad \mathsf{let}\ x = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{1}\ \widehat{2}](x_{\text{arg}}, \overline{1})\ \mathsf{in}\\
&\qquad \mathsf{let}\ x' = \langle X = A_{\text{env}},\\
&\qquad\quad \langle v_{\text{fix}}[Y_1]\ldots[Y_m], x_{\text{env}}\rangle\!:\!A_X\rangle\ \mathsf{in}\\
&\qquad \mathsf{let}\ x_0 = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{0}\ \widehat{k}](x_{\text{env}}, \overline{0})\ \mathsf{in}\ \ldots\\
&\qquad \mathsf{let}\ x_{k-1} = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{k-1}\ \widehat{k}](x_{\text{env}}, \overline{k-1})\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]\\
\mathcal{C}_{\text{exp}}[\![v_1[A_1,\ \ldots A_n](v_2)]\!] &= \mathsf{let}\ \langle X_{\text{env}},\ x_{\text{arg}}\rangle = \mathsf{open}\ \mathcal{C}_{\text{val}}[\![v_1]\!]\ \mathsf{in}\\
&\quad \mathsf{let}\ x_{\text{code}} = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{0}\ \widehat{2}](x_{\text{arg}}, \overline{0})\ \mathsf{in}\\
&\quad \mathsf{let}\ x_{\text{env}} = \mathsf{sel}[\mathsf{ltPrf}\ \widehat{1}\ \widehat{2}](x_{\text{arg}}, \overline{1})\ \mathsf{in}\\
&\quad x_{\text{code}}[A_1]\ldots[A_n]\ \langle x_{\text{env}}, \mathcal{C}_{\text{val}}[\![v_2]\!]\rangle\\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] &= \mathsf{let}\ x = \mathcal{C}_{\text{val}}[\![v]\!]\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]\\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = \mathsf{sel}[A](v, v')\ \mathsf{in}\ e]\!] &=\\
&\quad \mathsf{let}\ x = \mathsf{sel}[A](\mathcal{C}_{\text{val}}[\![v]\!], \mathcal{C}_{\text{val}}[\![v']\!])\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]\\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ \langle X,\ x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e]\!] &=\\
&\quad \mathsf{let}\ \langle X,\ x\rangle = \mathsf{open}\ \mathcal{C}_{\text{val}}[\![v]\!]\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]\\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = v_1 + v_2\ \mathsf{in}\ e]\!] &= \mathsf{let}\ x = \mathcal{C}_{\text{val}}[\![v_1]\!] + \mathcal{C}_{\text{val}}[\![v_2]\!]\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]\\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = v_1 < v_2\ \mathsf{in}\ e]\!] &= \mathsf{let}\ x = \mathcal{C}_{\text{val}}[\![v_1]\!] < \mathcal{C}_{\text{val}}[\![v_2]\!]\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]\\
\mathcal{C}_{\text{exp}}[\![\mathsf{if}[B, A](v,\ X_1.\, e_1,\ X_2.\, e_2)]\!] &=\\
&\quad \mathsf{if}[B, A](\mathcal{C}_{\text{val}}[\![v]\!],\ X_1.\, \mathcal{C}_{\text{exp}}[\![e_1]\!],\ X_2.\, \mathcal{C}_{\text{exp}}[\![e_2]\!])
\end{aligned}
$$

Figure 7: Closure conversion: from λ_K to λ_C.

Proposition 15 (Type Correctness of Closure Conversion) If $\cdot;\cdot \vdash_K v : A$, then $\cdot;\cdot \vdash_C \mathcal{C}_{\text{val}}[\![v]\!] : \mathsf{Cl}\ (A)\ \bot$.