A Type System for Certified Binaries

ZHONG SHAO and VALERY TRIFONOV
Yale University
BRATIN SAHA
Intel Corporation
and
NIKOLAOS PAPASPYROU
National Technical University of Athens

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.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Verification

Additional Key Words and Phrases: Certified code, proof-preserving compilation, typed intermediate languages

A preliminary version of this paper appeared in the Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02).

This work was supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, NSF ITR grant CCR-0081590, and NSF grant CCR-0208618. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

Authors' addresses: Zhong Shao and Valery Trifonov, Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520; email: {shao, trifonov}@cs.yale.edu; Bratin Saha, Microprocessor Technology Lab, Intel Corporation, Santa Clara, CA 95054; email: bratin.saha@intel.com; Nikolaos Papaspyrou, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, 15780 Zografou, Athens, Greece; email: nickie@softlab.ntua.gr.

1. INTRODUCTION

Proof-carrying code (PCC), as pioneered by Necula and Lee [1996] [Necula 1997], 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 [Necula 1998; Appel and Felten 2001]. For example, the Foundational PCC system [Appel and Felty 2000] can certify any property expressible in Church's higher-order logic. Harper [2000] and Burstall and McKinna [1991] call all these proof-carrying constructs certified binaries (or deliverables). 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. Most existing certifying compilers [Necula and Lee 1998; Colby et al. 2000] have focused on simple memory and control-flow safety only. Typed intermediate languages [Harper and Morrisett 1995] and typed assembly languages [Morrisett et al. 1998] are effective techniques for automatically generating certified code; however, none of these type systems can rival the expressiveness of the actual higher-order predicate logic (which could be used in any Foundational PCC system).

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 [Howard 1980] 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 [Paulin-Mohring 1993; Coquand and Huet 1988]) 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 and Geuvers 1999; Barendregt 1991] 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 [1999], we use singleton types [Hayashi 1991] 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 4.3). Xi and Pfenning [1999] 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 [Trifonov et al. 2000] 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 [Morrisett et al. 1998; Harper and Lillibridge 1993; Minamide et al. 1996; Barthe et al. 1999] 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).

—Our type language is a variant of the calculus of inductive constructions of Paulin-Mohring [1993] and Coquand and Huet [1988]. Following Werner [1994], 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 [Shao et al. 2001] 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 [Harper and Morrisett 1995; Xi and Pfenning 1999; Crary et al. 1999; Walker 2000; Crary and Weirich 2000; Trifonov et al. 2000] 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 for intensional type analysis [Harper and Morrisett 1995; Crary and Weirich 1999; Trifonov et al. 2000].

We have a prototype implementation of our new type system in the FLINT compiler [Shao 1997; Shao et al. 1998], 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 a 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 higher-order polymorphic λ-calculus F_ω [Girard 1972].

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 $\tau_1 \rightarrow \tau_2$ 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 "$\kappa \rightarrow \cdots \rightarrow \Omega$." Kind terms with other kind schemas, or type terms with other kinds are strictly referred to as "kind terms" or "type terms."

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) [Howard 1980] to

The type language:

$$(\textit{kscm}) \quad u ::= \mathsf{Kind} \mid \ldots$$
$$(\textit{kind}) \quad \kappa ::= \kappa_1 \rightarrow \kappa_2 \mid \Omega \mid \ldots$$
$$(\textit{type}) \quad \tau ::= t \mid \lambda t{:}\kappa.\,\tau \mid \tau_1\,\tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \forall t{:}\kappa.\,\tau \mid \ldots$$

The computation language:

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

Fig. 1. Typed λ-calculi—a skeleton

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. Good surveys of the previous work in this area can be found in Barendregt [1991] and Barendregt and Geuvers [1999].

2.2 Representing certifi ed 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 $\tau \rightarrow \Omega$. 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 $\mathsf{nat} \rightarrow \Omega$) 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 nat $\rightarrow$ 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:

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

The resulting certified binary has type

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

Here the type is not only dependent on values but also on function applications such as $f(x)$, so verifying the certified binary, which involves typechecking the proof, 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 nontermination. 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 [Sheldon and Gifford 1990] 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 [Girard 1972] 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 [Morrisett et al. 1998], but in the presence of dependent types, this is a very difficult problem [Barthe et al. 1999]. 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. But having dependent strong-sum and product types makes it harder to preserve this property, especially if the type-dependent values are first-class citizens (certified binaries are used to validate arbitrary data structures and program functions so they should be allowed to be passed as arguments, returned as results, or stored in memory).

Finally, supporting subset types in the presence of dependent strong-sum and product types is difficult if not impossible [Constable 1985; Nordstrom et al. 1990]. A certified binary of type Σx : nat. prime(x) contains a natural

number v and a proof that v is a prime. However, in many 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 computation term has to be compiled down to assembly code in any case, it is a bad idea to treat it as part of types. This 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 for proofs. The domain Nat is represented by a kind Nat; the predicate $Prime$ is represented by a dependent kind term Prime which maps a type term of kind Nat to 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 $\mathsf{Nat} \rightarrow \mathsf{Nat}$, 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:

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

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 [1999] and use singleton types [Hayashi 1991] to relate computation values to type terms. In the previous example, we introduce a singleton type constructor snat of kind $\mathsf{Nat} \rightarrow \Omega$. 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 [Crary et al. 1998], this certified binary is just an integer value of type $\mathsf{snat}(n)$ at run time.

Because there are strong separation between types and computation terms, a value v of type $\exists n : \mathsf{Nat}.\exists t : \mathsf{Prime}(n).\mathsf{snat}(n)$ is still implemented as a single integer at runtime thus achieving the effect of the subset type.

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 $\mathsf{Nat} \rightarrow \mathsf{Nat}$ to simulate the behavior of f (on all inputs where f does terminate) and show that f has the following type:

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

Here following Figure 1, we use $\forall$ and $\rightarrow$ 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 return, 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:

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

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.

It is important to understand the difference between typechecking and "type inference." The main objective of this paper is to develop a fully explicit framework where proofs and assertions can be used to certify programs that may contain side effects—the most important property is that typechecking (and proof-checking) in the new framework must be decidable. Type inference (*i.e.,* finding the proofs), on the other hand, could be undecidable: given an arbitrarily complex function f, we clearly cannot hope to automatically construct the corresponding g. In practice, however, it is often possible to first write down the specification g and then to write the corresponding program f. Carrying out this step and constructing the proof that f follows g is a challenging task, as in any other PCC system [Necula 1998; Appel and Felty 2000].

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 [Morrisett et al. 1998]. We also want to support intensional type analysis [Harper and Morrisett 1995] which is crucial for typechecking runtime services [Monnier et al. 2001].

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 [Trifonov et al. 2000] 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):

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

$$(\textit{kind}) \quad \kappa ::= \kappa_1 \rightarrow \kappa_2 \mid k \mid \Pi k\!:\!u.\, \kappa \mid \ldots$$

$$(\textit{type}) \quad \tau ::= t \mid \lambda t\!:\!\kappa.\, \tau \mid \tau_1\, \tau_2 \mid \lambda k\!:\!u.\, \tau \mid \tau[\kappa] \mid \ldots$$

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):

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

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 $\tau_1 \rightarrow \tau_2$ 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 $\kappa_1 \rightarrow \kappa_2$; we also add kind-level functions to represent predicates. Thus the type language naturally becomes the calculus of inductive constructions [Paulin-Mohring 1993].

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 nontrivial 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 [Barthe et al. 1999]. 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 [Morrisett et al. 1998; Harper and Lillibridge 1993; Minamide et al. 1996] all perform this 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 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 [Huet et al. 2000]. 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 [Paulin-Mohring 1989] 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. In practice these differences are fairly minor. The main objectives of this section is to give a quick introduction to the essential features in the Coq-like dependent type theory.

3.1 Motivations

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

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

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 $\Pi t : \kappa.\,\kappa'$ and $\kappa \to \kappa'$ are used to typecheck the type-level function $\lambda t : \kappa.\,\tau$ and the corresponding 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 the 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 [Trifonov et al. 2000]. It can also be used to define logic connectives and constants, as in

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

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 the corresponding application $\tau[u]$. This is not in the core calculus of constructions [Coquand and Huet 1988]. 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 $\lambda t : \kappa.\,\kappa'$ and the 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 for instance a binary relation:

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

so that LT t_1 t_2 is a proposition asserting that the natural number represented by t_1 is less than that of t_2.

$$\begin{array}{l}
\mathsf{Inductive\ Nat\ :\ Kind\ :=\ zero:Nat} \\
\qquad\qquad\qquad\qquad \mid \mathsf{succ:Nat\to Nat} \\
\\
\mathsf{plus\ :\ Nat\to Nat\to Nat} \\
\mathsf{plus(zero)} \ = \ \lambda t{:}\mathsf{Nat}.\, t \\
\mathsf{plus(succ}\ t) \ = \ \lambda t'{:}\mathsf{Nat.\ succ}\ (\mathsf{(plus}\ t)\ t') \\
\\
\mathsf{ifez\ :\ Nat\to(\Pi}k{:}\mathsf{Kind}.\ k\to(\mathsf{Nat}\to k)\to k) \\
\mathsf{ifez(zero)} \ = \ \lambda k{:}\mathsf{Kind}.\, \lambda t_1{:}k.\, \lambda t_2{:}\mathsf{Nat}\to k.\, t_1 \\
\mathsf{ifez(succ}\ t) \ = \ \lambda k{:}\mathsf{Kind}.\, \lambda t_1{:}k.\, \lambda t_2{:}\mathsf{Nat}\to k.\, t_2\ t
\end{array}$$

$$\begin{array}{l}
\mathsf{Inductive\ Bool\ :\ Kind\ :=\ true\ :Bool} \\
\qquad\qquad\qquad\qquad \mid \mathsf{false:Bool} \\
\\
\mathsf{le\ :\ Nat\to Nat\to Bool} \\
\mathsf{le(zero)} \ = \ \lambda t{:}\mathsf{Nat.\ true} \\
\mathsf{le(succ}\ t) \ = \ \lambda t'{:}\mathsf{Nat.\ ifez}\ t'\ \mathsf{Bool\ false\ (le}\ t) \\
\\
\mathsf{lt\ :\ Nat\to Nat\to Bool} \\
\mathsf{lt} \ = \ \lambda t{:}\mathsf{Nat.\ le\ (succ}\ t) \\
\\
\mathsf{Cond\ :\ Bool\to Kind\to Kind\to Kind} \\
\mathsf{Cond(true)} \ = \ \lambda k_1{:}\mathsf{Kind}.\, \lambda k_2{:}\mathsf{Kind}.\, k_1 \\
\mathsf{Cond(false)} \ = \ \lambda k_1{:}\mathsf{Kind}.\, \lambda k_2{:}\mathsf{Kind}.\, k_2
\end{array}$$

Fig. 2. Examples of inductive definitions and elimination

—Kind schemas $\Pi k : u.\, u'$ and $u \to u'$ are used to typecheck the kind-level function $\lambda k : u.\, \kappa$ and the 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:

$$\mathsf{Not\ :\ Kind\to Kind} \ = \ \lambda k{:}\mathsf{Kind}.\, k\to\mathsf{False}$$

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 for defining inductive types [Paulin-Mohring 1993]. The term $\mathsf{Ind}(k : \mathsf{Kind})\{\vec{\kappa}\}$ introduces an inductive kind k with constructors whose kinds are listed in $\vec{\kappa}$. Here k must only occur "positively" inside each κ_i (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 [Werner 1994; Paulin-Mohring 1993].

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

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

Fig. 3. Syntax of the type language TL

schema u as the result. These iterators generalize the Typerec operator used in intensional type analysis [Harper and Morrisett 1995; Crary and Weirich 1999; Trifonov et al. 2000].

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 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:

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

The general ι-reduction rule is defined formally in Appendix A. In our examples, we take the liberty of using the pattern-matching 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, is expanded into TL using large elimination on Bool, of the form $\mathsf{Elim}[\mathsf{Bool}, u](\tau)\{\kappa_1; \kappa_2\}$, where τ is of kind Bool, and both the true and false branches (κ_1 and κ_2) have kind schema u.

3.2 Formalization

We want to give a formal semantics to TL and then reason about its metatheoretic properties. But the five Π constructs have many similarities, so in the rest of this paper, we will model TL as a pure type system (PTS) [Barendregt 1991] extended with inductive definitions. Intuitively, instead of having a separate syntactic 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 specific.

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

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

Here $\mathcal{S}$ is the set of emphsorts used to denote universes. We have added the constant Ext to support quantification over Kscm. The names we use for sorts reflect the fact that we have lifted the language one level up; they are related to other systems via the following table:

System	Notation		
TL	Kind	Kscm	Ext
Werner [1994]	Set	Type	Ext
Coq/CIC [Huet et al. 2000]	Set, Prop	Type(0)	Type(1)
Barendregt [1991]	$*$	$\Box$	$\triangle$

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 pairs (*rules*) in the set $\mathcal{R}$ are used to define the 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:

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

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

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

The typing judgment for TL in PTS style 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:

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

Taking rule $(\mathsf{Kind}, \mathsf{Kscm})$ 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 λ-definitions and applications:

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

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

The reduction relations can also be shared. TL supports the standard β- and η-reductions (denoted by $\rightsquigarrow_\beta$ and $\rightsquigarrow_\eta$) plus the previously mentioned ι-reduction (denoted by $\rightsquigarrow_\iota$) on inductive objects (see Appendix A). The relations $\rhd_\beta$, $\rhd_\eta$, and $\rhd_\iota$ are the contextual closures of 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, and 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 [1994] and Geuvers [1993], 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 [Shao et al. 2001].

Theorem 3.1 (Subject reduction) If the judgment $\Delta \vdash A : B$ is derivable, and $A \rhd A'$, then $\Delta \vdash A' : B$ is derivable.

Proof sketch The detailed proof is given in the companion technical report [Shao et al. 2001]. We first define a calculus of *unmarked terms*. These are TL terms with no annotations at lambda abstractions. We show that this language is confluent. From this, we can prove that TL satisfies a weak form of confluence (also known as *the Geuvers lemma* [Geuvers 1993]); it says that a term that is equal to one in head normal form can be reduced to an η-expanded version of this head normal form. From the weak confluence, we then prove the inversion lemma which relates the structure of a term to its typing derivation. We then prove the uniqueness of types and subject reduction for $\beta\iota$ reductions. Finally, we prove the strengthening lemma and then subject reduction for η reduction. □

Theorem 3.2 (Strong normalization) All well typed terms are strongly normalizing.

Proof sketch The detailed proof is presented in our technical report [Shao et al. 2001]. It is a straightforward extension of the proof given by Werner [1994]. First we introduce a calculus of *pure terms*; this is just the pure λ-calculus extended with a recursive filtering operator; we do this so that we can operate in a confluent calculus. We then define a notion of reducibility candidates; every kind schema gives rise to a reducibility candidate; we also

show how these candidates can be constructed inductively. We define a notion of well constructed kinds which is a weak form of typing. We associate an interpretation to each well formed kind. We show that under adequate conditions, this interpretation is a candidate. We show that type level constructs such as abstractions and constructors belong to the candidate associated with their kind. We show that the interpretation of a kind remains the same under $\beta\eta$ reduction. We then define a notion of kinds that are invariant on their domain—these are kinds whose interpretation remains the same upon reduction. We show that kinds formed with large elimination are invariant on their domain. From here we can show the strong normalization of the calculus of pure terms; we show that if a type is well formed, then the pure term derived from it is strongly normalizing. Finally, we reduce the strong normalization of all well formed terms to the strong normalization of pure terms. □

Theorem 3.3 (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 $A = A'$.

Proof sketch The detailed proof is given in the companion technical report [Shao et al. 2001]. We first prove that a well typed term in $\beta\iota$ normal form has the same η reductions as its corresponding unmarked term. From here, we know that if A and A' are in normal form, then their corresponding unmarked terms are equal. We then show that the annotations in the λ-abstractions are equal. □

Theorem 3.4 (Consistency of the logic) There exists no term A for which $\cdot \vdash A : \mathsf{False}$.

Proof sketch Suppose A is a term for which $\cdot \vdash A : \mathsf{False}$. By Theorem 3.2, there exists a normal form B for A. By Theorem 3.1 $\cdot \vdash B : \mathsf{False}$. We can show now that this leads to a contradiction by case analysis of the possible normal forms of types in the calculus. □

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, for instance type safety. This appears to be even more promising for a system with a type language like CIC, which is

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

Fig. 4. Syntax of the computation language λ_H.

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 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 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 4.2, 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.

4.1 Dynamic semantics

We present a small step call-by-value operational semantics for λ_H in the style of Wright and Felleisen [1994]. The values are defined inductively by

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

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

$$
\begin{array}{rcll}
(\lambda x{:}A.\, e)\ v & \hookrightarrow & [v/x]e & \text{(R-}\beta\text{)} \\
(\Lambda X{:}B.\, f)[A] & \hookrightarrow & [A/X]f & \text{(R-TY-}\beta\text{)} \\
\mathsf{sel}[A](\langle v_0, \ldots v_{n-1}\rangle, \overline{m}) & \hookrightarrow & v_m \qquad (m < n) & \text{(R-SEL)} \\
\mathsf{open}\ \langle X' = A,\, v{:}A'\rangle\ \mathsf{as}\ \langle X,\, x\rangle\ \mathsf{in}\ e & \hookrightarrow & [v/x][A/X]e & \text{(R-OPEN)} \\
(\mathsf{fix}\ x{:}A.\, f)\ v & \hookrightarrow & ([\mathsf{fix}\ x{:}A.\, f/x]f)\ v & \text{(R-FIX)} \\
(\mathsf{fix}\ x{:}A.\, f)[A'] & \hookrightarrow & ([\mathsf{fix}\ x{:}A.\, f/x]f)[A'] & \text{(R-TYFIX)} \\
\overline{m} + \overline{n} & \hookrightarrow & \overline{m+n} & \text{(R-ADD)} \\
\overline{m} < \overline{n} & \hookrightarrow & \mathsf{tt} \qquad (m < n) & \text{(R-LT-T)} \\
\overline{m} < \overline{n} & \hookrightarrow & \mathsf{ff} \qquad (m \geq n) & \text{(R-LT-F)} \\
\mathsf{if}\ [B, A](\mathsf{tt},\, X_1.\, e_1,\, X_2.\, e_2) & \hookrightarrow & [A/X_1]e_1 & \text{(R-IF-T)} \\
\mathsf{if}\ [B, A](\mathsf{ff},\, X_1.\, e_1,\, X_2.\, e_2) & \hookrightarrow & [A/X_2]e_2 & \text{(R-IF-F)}
\end{array}
$$

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

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

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.

4.2 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.

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

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

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

and often drop the sort s when $s = \mathsf{Kind}$; for example 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

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

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 $\mathsf{snat}\ (\mathsf{succ}\ \mathsf{zero})$ 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 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 assign only 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

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

$$\frac{\Delta \vdash \mathsf{Kind} : \mathsf{Kscm}}{\Delta \vdash \cdot \ ok} \ (\textsc{te-mt}) \qquad \frac{\Delta \vdash \Gamma \ ok \quad \Delta \vdash A : \Omega}{\Delta \vdash \Gamma, x{:}A \ ok} \ (\textsc{te-ext})$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash x : \Gamma(x)} \ (\textsc{e-var}) \qquad \frac{\Delta; \Gamma \vdash e : \mathsf{snat}\ A \quad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A'}{\Delta; \Gamma \vdash e + e' : \mathsf{snat}\ (\mathsf{plus}\ A\ A')} \ (\textsc{e-add})$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \overline{n} : \mathsf{snat}\ \widehat{n}} \ (\textsc{e-nat}) \qquad \frac{\Delta; \Gamma \vdash e : \mathsf{snat}\ A \quad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A'}{\Delta; \Gamma \vdash e < e' : \mathsf{sbool}\ (\mathsf{lt}\ A\ A')} \ (\textsc{e-lt})$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \mathsf{tt} : \mathsf{sbool}\ \mathsf{true}} \ (\textsc{e-true}) \qquad \frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \mathsf{ff} : \mathsf{sbool}\ \mathsf{false}} \ (\textsc{e-false})$$

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

$$\frac{\Delta \vdash A : \Omega \quad \Delta; \Gamma, x{:}A \vdash f : A}{\Delta; \Gamma \vdash \mathsf{fix}\ x{:}A.\, f : A} \ (\textsc{e-fix}) \qquad \frac{\Delta; \Gamma \vdash e_1 : A \to A' \quad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : A'} \ (\textsc{e-app})$$

$$\frac{\Delta \vdash A : \Omega \quad \Delta; \Gamma, x{:}A \vdash e : A'}{\Delta; \Gamma \vdash \lambda x{:}A.\, e : A \to A'} \ (\textsc{e-fun})$$

$$\frac{\Delta \vdash B : s \quad \Delta, X{:}B; \Gamma \vdash f : A}{\Delta; \Gamma \vdash \Lambda X{:}B.\, f : \forall_s X{:}B.\, A} \ (\textsc{e-tfun}) \quad \text{where } X \notin \Delta,\ s \neq \mathsf{Ext}$$

$$\frac{\Delta; \Gamma \vdash e : \forall_s\ B\ A \quad \Delta \vdash A' : B}{\Delta; \Gamma \vdash e[A'] : A\ A'} \ (\textsc{e-tapp}) \quad \text{where } s \neq \mathsf{Ext}$$

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

$$\frac{\Delta; \Gamma \vdash e : \exists_s\ B\ A \quad \Delta \vdash A' : \Omega \quad \Delta, X{:}B; \Gamma, x{:}A\ X \vdash e' : A'}{\Delta; \Gamma \vdash \mathsf{open}\ e\ \mathsf{as}\ \langle X,\ x \rangle\ \mathsf{in}\ e' : A'} \ (\textsc{e-open}) \quad \text{where } X \notin \Delta,\ s \neq \mathsf{Ext}$$

$$\frac{\textit{for all } i < n \quad \Delta; \Gamma \vdash e_i : A_i}{\Delta; \Gamma \vdash \langle e_0, \ldots e_{n-1} \rangle : \mathsf{tup}\ \widehat{n}\ (\mathsf{nth}\ (A_0 :: \ldots :: A_{n-1} :: \mathsf{nil}))} \ (\textsc{e-tup})$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{tup}\ A''\ B \quad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A' \quad \Delta \vdash A : \mathsf{LT}\ A'\ A''}{\Delta; \Gamma \vdash \mathsf{sel}[A](e, e') : B\ A'} \ (\textsc{e-sel})$$

$$\frac{\Delta; \Gamma \vdash e : A \quad A =_{\beta\eta\iota} A' \quad \Delta \vdash A' : \Omega}{\Delta; \Gamma \vdash e : A'} \ (\textsc{e-conv})$$

Fig. 5. Static semantics of the computation language λ_H.

Rule E-TUP assigns to a tuple a type of the form 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:

$$\begin{array}{l}\mathsf{Inductive\ List\ :\ Kind\ :=\ nil : List \mid cons : \Omega \to List \to List} \\ \mathsf{nth\ :\ List \to Nat \to \Omega} \\ \mathsf{nth\ nil} = \lambda t{:}\mathsf{Nat.\ void} \\ \mathsf{nth\ (cons}\ t_1\ t_2) = \lambda t{:}\mathsf{Nat.\ ifez}\ t\ \Omega\ t_1\ (\mathsf{nth}\ t_2)\end{array}$$

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 $\forall m, n \in \mathbb{N}.\, m < n \supset m{+}1 < n{+}1$:

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

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 (given in Section 4.3); this is sufficient since we never analyze proof objects, so the full power of elimination is unnecessary for our use of 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 conditional $\mathsf{if}\ [B, A](e,\ X_1.\, e_1,\ X_2.\, e_2)$ allows information obtained dynamically (*e.g.,* through comparisons) to be made available for static reasoning in the form of proof parameters to its branches. The type term A represents a proof of the proposition encoded by either $B\ \mathsf{true}$ or $B\ \mathsf{false}$, depending on the value of e. This proof is bound to the type variable (X_1 or X_2) of the appropriate branch, which can use it in the construction of other proofs, or with a proof-consuming primitive like sel. The correspondence between the value of e and the kind of A is again established through a singleton boolean type. Thus for instance if the run-time value of e asserts the truthfulness of some proposition P, since the type parameter A'' of the singleton type of e reflects the value of e at the type level, we can define B so that $B\ A''$ represents P or $\neg P$, depending on whether $A'' =_{\beta\eta\iota} \mathsf{true}$ or $A'' =_{\beta\eta\iota} \mathsf{false}$, and reason in each of the two branches under the assumption that P or $\neg P$, respectively. Of course, for this reasoning to be sound, we need a proof that A'' indeed reflects the truthfulness of P, that is, we need a proof term A of kind $B\ A''$.

In fact if is more flexible than that, because $B\ \mathsf{false}$ does not have to be the negation of $B\ \mathsf{true}$, one can have imprecise information flow into the branches. In particular the encoding of the usual oblivious (in proof-passing sense) if is possible using $B = \lambda t : \mathsf{Bool}.\, \mathsf{True}$; section 4.3 gives another example, where the information is precise only in one branch of the conditional.

4.3 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 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

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

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

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

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 below; as its kind suggests, ltPrf A A' evaluates to a proof of LT A A', if A and A' represent natural numbers n and n' such that $n < n'$.

The comparison i′ < 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 ltPrf t' t represents a proof of the corresponding proposition at the type level, namely 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 4.6) guarantees that implementations of sel need not check the subscript at runtime. Since the proof t_2 is ignored in the "else" branch, 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

$$\begin{aligned}&\mathsf{vector} \;:\; \Omega \to \Omega\\ &\mathsf{vector} \;=\; \lambda t{:}\Omega.\, \exists t'{:}\mathsf{Nat}.\, \mathsf{snat}\ t' \times \mathsf{vec}\ t'\ t\end{aligned}$$

Now we can write a wrapper function for sumVec operating on packaged vectors.

$$\begin{aligned}&\mathsf{sumVector} : \mathsf{vector\ nat} \to \mathsf{nat}\\ &\quad \equiv \lambda \mathsf{v} : \mathsf{vector\ nat}.\\ &\qquad \mathsf{open\ v\ as}\ \langle t', \mathsf{v}'\rangle\ \mathsf{in}\ \mathsf{sumVec}[t']\ (\mathsf{sel}[\mathsf{ltPrf}\ \widehat{0}\ \widehat{2}](\mathsf{v}', \overline{0}))\ (\mathsf{sel}[\mathsf{ltPrf}\ \widehat{1}\ \widehat{2}](\mathsf{v}', \overline{1}))\end{aligned}$$

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

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

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

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

Next we define dependent conditionals on kinds Nat and Bool.

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

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

Note that, unlike the examples in Figure 2, the types of the branches in each of these definitions are different: The type of the true branch of dep_if is

$$\Pi k : \mathsf{Bool} \to \mathsf{Kind}.\ k\ \mathsf{true} \to k\ \mathsf{false} \to k\ \mathsf{true},$$

while that of its false branch is

$$\Pi k : \mathsf{Bool} \to \mathsf{Kind}.\ k\ \mathsf{true} \to k\ \mathsf{false} \to k\ \mathsf{false}.$$

This is achieved by specifying the kind term

$$\lambda t : \mathsf{Bool}.\ \Pi k : \mathsf{Bool} \to \mathsf{Kind}.\ k\ \mathsf{true} \to k\ \mathsf{false} \to k\ t$$

as the second parameter of the Elim construct for which the sugared definition of dep_if above stands. The resulting elimination term is type-correct because the type of each branch is obtained by applying this kind term to the corresponding constructor of Bool.

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

```
LTcond : Nat → Nat → Kind
LTcond = λt' : Nat. λt : Nat. LTOrTrue t' t (lt t' t)

LTsucc : Nat → Nat → Bool → Kind
LTsucc = λt' : Nat. λt : Nat. λt'' : Bool.
           LTOrTrue t' t t'' → LTOrTrue (succ t') (succ t) t''


ltPrf : Πt' : Nat. Πt : Nat. LTcond t' t
ltPrf = λt' : Nat.
  Elim[Nat, λt'_1 : Nat. Πt_1 : Nat. LTcond t'_1 t_1](t'){
    λt_1 : Nat. dep_ifez t_1 (LTcond zero) id ltzs;
    λt'_1 : Nat. λt_P : (Πt_1 : Nat. LTcond t'_1 t_1). λt_1 : Nat.
      dep_ifez
        t_1
        (LTcond (succ t'_1))
        id
        (λt_1 : Nat. dep_if (lt t'_1 t_1) (LTsucc t'_1 t_1) (ltss t'_1 t_1) (id True) (t_P t_1))}
```

4.4 Example: Type conversions

The language λ_H offers only the bare minimum of constructs for programming with TL types. However the reader may recall that λ_H is an intermediate language, and ease of programming in it is not necessarily of high importance. Much more important is that it has the flexibility to express the more complex relationships between terms and types in other languages, to do this in terms of simple constructs, which are relatively simple to reason about and transform, and do it at no run-time cost. To a large extent this flexibility comes from the use of type-level proof terms in λ_H.

One example of the power of programming with proof terms is the ability to use λ_H in a way which allows more general type conversions than those permitted by rule E-CONV. This rule allows the conversion of a term's type only to other $\beta\eta\iota$-equivalent types, but not to types which are provably equivalent in some weaker sense. For instance it is impossible to convert a λ_H-term of type vec (plus t_1 t_2) nat to a term of type vec (plus t_2 t_1) nat in a context where the distinct type variables t_1 and t_2 have kind Nat, because the type terms plus t_1 t_2 and plus t_2 t_1, being different normal forms, are not $\beta\eta\iota$-equivalent.

A solution is to instead define and use types which represent equivalence classes with respect to a relation of interest, in this case raw datatypes of λ_H packaged together with proof terms of type equivalence. When a parameter of a type constructor must be subjected to conversions in our program, we can replace it by a derived type constructor which hides the actual "value"

of this parameter, and exposes only an equivalent value, with a proof of their equivalence hidden in the package. Thus the singleton integer type $\mathsf{snat}(A)$ can be replaced by the type $\mathsf{snatp}(A)$, defined as follows:

$$\begin{aligned}&\mathsf{snatp} \;:\; \mathsf{Nat} \to \Omega\\ &\mathsf{snatp} = \lambda t' : \mathsf{Nat}.\; \exists t : \mathsf{Nat}.\; \exists \mathsf{P} : \mathsf{Eq}\ \mathsf{Nat}\ t'\ t.\; \mathsf{snat}(t)\end{aligned}$$

In a package of type $\mathsf{snatp}(A)$ the variable P is bound to a proof of the equality between A and the witness type bound to t, which represents the actual value of the term-level integer component. As we will show shortly, this allows to easily convert a term of type $\mathsf{snatp}(A)$ to type $\mathsf{snatp}(A')$ when A and A' represent natural numbers provably equal in the given context. The kind of equality proofs Eq can be defined in CIC following Paulin-Mohring [1993] as

$$\begin{aligned}&\mathsf{Eq} \;:\; \Pi k : \mathsf{Kind}.\; k \to k \to \mathsf{Kind}\\ &\mathsf{Eq} = \lambda k : \mathsf{Kind}.\; \lambda t : k.\; \mathsf{Ind}(k' : k \to \mathsf{Kind})\{k'\ t\}\\ \\ &\mathsf{refl} \;:\; \Pi k : \mathsf{Kind}.\; \Pi t : k.\; \mathsf{Eq}\ k\ t\ t\\ &\mathsf{refl} = \lambda k : \mathsf{Kind}.\; \lambda t : k.\; \mathsf{Ctor}\,(1, \mathsf{Eq}\ k\ t)\end{aligned}$$

and its elimination allows us to define a type term showing this is actually Leibniz equality:

$$\mathsf{Leibniz} : \Pi k : \mathsf{Kind}.\; \Pi t : k.\; \Pi t' : k.\; \mathsf{Eq}\ k\ t\ t' \to \Pi \mathsf{P} : k \to \mathsf{Kind}.\; \mathsf{P}\ t \to \mathsf{P}\ t'$$

By this definition of equality, the normal form of a term representing a proof of equality between closed types A and A' is an application of the constructor refl, whose kind ensures that the types are $\beta\eta\iota$-equivalent. The expressiveness comes from the possibility to construct proofs of equality using case analysis with dependent elimination to relate different normal forms. Consider the following example. Proving that zero is a left unit of plus is trivial:

$$\begin{aligned}&\mathsf{leftUnit} \;:\; \Pi t : \mathsf{Nat}.\; \mathsf{Eq}\ \mathsf{Nat}\ t\ (\mathsf{plus}\ \mathsf{zero}\ t)\\ &\mathsf{leftUnit} = \mathsf{refl}\ \mathsf{Nat}\end{aligned}$$

because according to our definition of plus we have $\mathsf{plus}\ \mathsf{zero}\ t \rhd t$. Not so with proving that zero is a right unit of plus: The type term $\mathsf{plus}\ t\ \mathsf{zero}$ is in normal form (assuming plus stands for the elimination term of TL defined in user-friendly form in Figure 2), not convertible to t. However it is possible to encode an inductive proof, using dependent elimination on Nat:

$$\begin{aligned}&\mathsf{rightUnit} \;:\; \Pi t : \mathsf{Nat}.\; \mathsf{Eq}\ \mathsf{Nat}\ t\ (\mathsf{plus}\ t\ \mathsf{zero})\\ &\mathsf{rightUnit}\ \mathsf{zero} \quad\;\; = \mathsf{refl}\ \mathsf{Nat}\ \mathsf{zero}\\ &\mathsf{rightUnit}\ (\mathsf{succ}\ t) = \mathsf{eqf}\ \mathsf{Nat}\ \mathsf{Nat}\ \mathsf{succ}\ t\ (\mathsf{plus}\ t\ \mathsf{zero})\ (\mathsf{rightUnit}\ t)\end{aligned}$$

where

$$\begin{aligned}&\mathsf{eqf} \;:\; \Pi k : \mathsf{Kind}.\; \Pi k' : \mathsf{Kind}.\; \Pi \mathsf{f} : k \to k'.\; \Pi t : k.\; \Pi t' : k.\; \mathsf{Eq}\ k\ t\ t' \to \mathsf{Eq}\ k'\ (\mathsf{f}\ t)\ (\mathsf{f}\ t')\\ &\mathsf{eqf} = \lambda k : \mathsf{Kind}.\; \lambda k' : \mathsf{Kind}.\; \lambda \mathsf{f} : k \to k'.\; \lambda t : k.\; \lambda t' : k.\; \lambda \mathsf{p} : \mathsf{Eq}\ k\ t\ t'.\\ &\qquad \mathsf{Leibniz}\ k\ t\ t'\ \mathsf{p}\ (\lambda t'' : k.\; \mathsf{Eq}\ k'\ (\mathsf{f}\ t)\ (\mathsf{f}\ t''))\ (\mathsf{refl}\ k'\ (\mathsf{f}\ t))\end{aligned}$$

The type term eqf constructs a proof of equality between the results of two applications of a function, given a proof of equality between the arguments. In

rightUnit it is employed to obtain from the inductive hypothesis (with proof represented by rightUnit t) a proof of Eq Nat (succ t) (succ (plus t zero)), which by the definition of plus is $\beta\eta\iota$-equivalent to the goal Eq Nat (succ t) (plus (succ t) zero). The dependency between the parameter of rightUnit and the types of the right-hand side branches must be specified using $\lambda t : \mathsf{Nat}.\ \mathsf{Eq\ Nat}\ t\ (\mathsf{plus}\ t\ \mathsf{zero})$ as the second parameter of the Elim term in the unsugared TL definition of rightUnit; the type of the zero branch is $\beta\eta\iota$-equivalent to Eq Nat zero (plus zero zero), and that of the succ branch with parameter t is Eq Nat (succ t) (plus (succ t) zero).

Returning to type conversions in λ_H, suppose now that we have a vector of length plus $t_1\ t_2$, while a function we want to apply to it expects a vector of length plus $t_2\ t_1$. Let us define the proof-augmented version of the vector type as follows.

$$\begin{aligned}&\mathsf{vecp}\ :\ \mathsf{Nat} \to \Omega \to \Omega\\ &\mathsf{vecp} = \lambda t' : \mathsf{Nat}.\ \lambda t_1 : \Omega.\ \exists t : \mathsf{Nat}.\ \exists \mathsf{P} : \mathsf{Eq\ Nat}\ t'\ t.\ \mathsf{vec}\ t\ t_1\end{aligned}$$

The "old" vectors can be trivially converted to the new type by giving them the same size they had: If v_1 has type vec $A\ B$, then

$$\begin{aligned}&\langle t = A,\ \langle \mathsf{P} = \mathsf{refl\ Nat}\ A,\ \mathsf{v}_1 : \mathsf{vec}\ A\ B\rangle\\ &\qquad :\ \exists \mathsf{P} : \mathsf{Eq\ Nat}\ A\ t.\ \mathsf{vec}\ t\ B\rangle\end{aligned}$$

has type vecp $A\ B$. Selection from these vectors can be performed for the same index expressions as for the corresponding "old" vectors—constructing a proof of LT $A'\ t$ from proofs of LT $A'\ A$ and Eq Nat $A\ t$ is straightforward. Conversion of the type of some term v from vecp (plus $t_1\ t_2$) nat to vecp (plus $t_2\ t_1$) nat is performed by the expression

$$\begin{aligned}&\mathsf{open\ v\ as}\ \langle t,\ \mathsf{v}'\rangle\ \mathsf{in\ open\ v}'\ \mathsf{as}\ \langle \mathsf{P},\ \mathsf{v}''\rangle\ \mathsf{in}\\ &\quad \langle t = t,\\ &\quad\ \langle \mathsf{P} = \mathsf{eqTrans\ Nat}\ (\mathsf{plus}\ t_2\ t_1)\ (\mathsf{plus}\ t_1\ t_2)\ t\ (\mathsf{plusSym}\ t_2\ t_1)\ \mathsf{P},\\ &\quad\ \mathsf{v}'' : \mathsf{vec}\ t\ \mathsf{nat}\rangle\\ &\quad\ :\ \exists \mathsf{P} : \mathsf{Eq\ Nat}\ (\mathsf{plus}\ t_2\ t_1)\ t.\ \mathsf{vec}\ t\ \mathsf{nat}\rangle\end{aligned}$$

where eqTrans is a proof of the transitivity of equality

$$\begin{aligned}&\mathsf{eqTrans}\ :\ \Pi k : \mathsf{Kind}.\ \Pi t : k.\ \Pi t' : k.\ \Pi t'' : k.\ \mathsf{Eq}\ k\ t\ t' \to \mathsf{Eq}\ k\ t'\ t'' \to \mathsf{Eq}\ k\ t\ t''\\ &\mathsf{eqTrans} = \lambda k : \mathsf{Kind}.\ \lambda t : k.\ \lambda t' : k.\ \lambda t'' : k.\ \lambda \mathsf{p} : \mathsf{Eq}\ k\ t\ t'.\\ &\qquad\qquad \lambda \mathsf{p}' : \mathsf{Eq}\ k\ t'\ t''.\ \mathsf{Leibniz}\ k\ t'\ t''\ \mathsf{p}'\ (\mathsf{Eq}\ k\ t)\ \mathsf{p}\end{aligned}$$

and plusSym is a proof of the symmetry of plus (using the lemma succPlus proving that $\forall n, m \in \mathbb{N}.\ (n + m) + 1 = n + (m + 1)$):

$$\begin{aligned}&\mathsf{plusSym}\ :\ \Pi t : \mathsf{Nat}.\ \Pi t' : \mathsf{Nat}.\ \mathsf{Eq\ Nat}\ (\mathsf{plus}\ t\ t')\ (\mathsf{plus}\ t'\ t)\\ &\mathsf{plusSym\ zero}\quad\ = \mathsf{rightUnit}\\ &\mathsf{plusSym}\ (\mathsf{succ}\ t) = \lambda t' : \mathsf{Nat}.\ \mathsf{eqTrans\ Nat}\\ &\qquad\qquad\qquad\qquad (\mathsf{plus}\ (\mathsf{succ}\ t)\ t')\\ &\qquad\qquad\qquad\qquad (\mathsf{succ}\ (\mathsf{plus}\ t'\ t))\\ &\qquad\qquad\qquad\qquad (\mathsf{plus}\ t'\ (\mathsf{succ}\ t))\\ &\qquad\qquad\qquad\qquad (\mathsf{eqf\ Nat\ Nat\ succ}\ (\mathsf{plus}\ t\ t')\ (\mathsf{plus}\ t'\ t)\ (\mathsf{plusSym}\ t\ t'))\\ &\qquad\qquad\qquad\qquad (\mathsf{succPlus}\ t'\ t)\end{aligned}$$

$$
\begin{array}{l}
\mathsf{succPlus} : \Pi t\!:\!\mathsf{Nat}.\,\Pi t'\!:\!\mathsf{Nat}.\,\mathsf{Eq\ Nat}\ (\mathsf{succ}\ (\mathsf{plus}\ t\ t'))\ (\mathsf{plus}\ t\ (\mathsf{succ}\ t')) \\
\mathsf{succPlus\ zero} \quad = \lambda t'\!:\!\mathsf{Nat}.\,\mathsf{refl\ Nat}\ (\mathsf{succ}\ t') \\
\mathsf{succPlus}\ (\mathsf{succ}\ t) = \lambda t'\!:\!\mathsf{Nat}.\,\mathsf{eqf\ Nat\ Nat\ succ} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{succ}\ (\mathsf{plus}\ t\ t')) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{plus}\ t\ (\mathsf{succ}\ t')) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{succPlus}\ t\ t')
\end{array}
$$

Similar proof terms can be found, among many other, in standard proof libraries (*e.g.,* that of Coq [Huet et al. 2000]).

Due to the explicit use of proof terms, this technique for support of type conversions can also exploit equivalences which are valid only locally, for instance in a branch of a term-level conditional. To simplify the following example, let us extend the computation language with a comparison for equality between natural numbers with the obvious semantics.[1] In the following example, two vectors of unrelated (in general) sizes can be converted to the same type if they are dynamically determined to have the same size.

$$
\begin{array}{l}
\Lambda t\!:\!\mathsf{Nat}.\,\lambda \mathsf{n}\!:\!\mathsf{snat}(t).\,\lambda \mathsf{v}\!:\!\mathsf{vecp}\ t\ \mathsf{nat}. \\
\Lambda t'\!:\!\mathsf{Nat}.\,\lambda \mathsf{n}'\!:\!\mathsf{snat}(t').\,\lambda \mathsf{v}'\!:\!\mathsf{vecp}\ t'\ \mathsf{nat}. \\
\ \mathsf{if}\ [\mathsf{EqOrTrue}\ t\ t', \mathsf{eqPrf}\ t\ t'] \\
\quad (\mathsf{n} = \mathsf{n}', \\
\quad\ \mathsf{P}.\ \ldots\,\mathsf{open}\ \mathsf{v}'\ \mathsf{as}\ \langle t_1,\ \mathsf{x}\rangle\ \mathsf{in\ open\ x\ as}\ \langle \mathsf{P}_1,\ \mathsf{y}\rangle\ \mathsf{in} \\
\qquad\qquad \langle t_2 = t_1,\ \langle \mathsf{P}_2 = \mathsf{eqTrans\ Nat}\ t\ t'\ t_1\ \mathsf{P}\ \mathsf{P}_1,\ \mathsf{y}\!:\!\mathsf{vec}\ t_1\ \mathsf{nat}\rangle,\ \ldots \\
\qquad\qquad\qquad\quad :\ \ \exists \mathsf{P}_2\!:\!\mathsf{Eq\ Nat}\ t\ t_2.\,\mathsf{vec}\ t_2\ \mathsf{nat}\rangle \\
\quad \_.\ \ldots)
\end{array}
$$

where EqOrTrue and eqPrf are the analogues of LTOrTrue and ltPrf from Section 4.3. The proof of Eq Nat $t\ t_2$, bound to P_2, is constructed by transitivity from the proof of Eq Nat $t\ t'$, bound to P by the conditional, and the proof of Eq Nat $t'\ t_2$, extracted from the package v′ and bound to P_1. As a result the type of the open term, which is a repackaged v′, is vecp t nat—the type of v.

Notice that all terms involved in the type conversions have no computational overhead and will be eliminated under type-erasure semantics; we emphasized this fact in the examples by placing the conversions inline.

As with the kind term LT, strictly speaking TL does not allow the above definition of Eq, but its Church encoding has the same properties for our purposes, since we do not need dependent or large elimination of equality proof terms for the proof compositions shown here. The Church encoding of the equality kind, its "constructor," and its elimination are as follows.

$$
\begin{array}{ll}
\mathsf{Eq} & = \lambda k\!:\!\mathsf{Kind}.\,\lambda t\!:\!k.\,\lambda t'\!:\!k.\,\Pi \mathsf{P}\!:\!k \to \mathsf{Kind}.\,\mathsf{P}\ t \to \mathsf{P}\ t' \\
\mathsf{refl} & = \lambda k\!:\!\mathsf{Kind}.\,\lambda t\!:\!k.\,\lambda \mathsf{P}\!:\!k \to \mathsf{Kind}.\,\lambda \mathsf{p}\!:\!\mathsf{P}\ t.\,\mathsf{p} \\
\mathsf{Leibniz} & = \lambda k\!:\!\mathsf{Kind}.\,\lambda t\!:\!k.\,\lambda t'\!:\!k.\,\lambda \mathsf{p}\!:\!\mathsf{Eq}\ k\ t\ t'.\,\mathsf{p}
\end{array}
$$

[1] Comparison for equality can be derived from the less-than comparison of λ_H; we will also need a straightforward to define proof term for $\Pi t\!:\!\mathsf{Nat}.\,\Pi t'\!:\!\mathsf{Nat}.\,\mathsf{Not}\ (\mathsf{LT}\ t\ t') \to \mathsf{Not}\ (\mathsf{LT}\ t'\ t) \to \mathsf{Eq\ Nat}\ t\ t'$ or equivalent.

Clearly there are opportunities to generalize this style to weaker relations of equivalence, which reveal partial information about the hidden type parameters. We will not explore this topic here.

4.5 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 4.3) 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 (metalogical) proof of the side condition $m < n$, required by rule R-SEL. Similarly, subject reduction (Lemma 4.5) 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 4.1 (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

1: By induction on m and inspection of the definition of plus.
2: By induction on m and the definition of le (Figure 2); for the forward direction the auxiliary inductive hypothesis is that for all n, if $\mathsf{le}\ \widehat{m}\ \widehat{n}$, then $m \leq n$.
3: For the forward direction it suffices to observe that the structure of the metalogical 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. □

We also need a guarantee that the equivalence of constructor applications implies the equivalence of the constructors and their arguments.

Lemma 4.2 If $\mathsf{Ctor}\ (i, I)\ \vec{A} =_{\beta\eta\iota} \mathsf{Ctor}\ (i', I')\ \vec{A}'$, then $i = i'$, $I =_{\beta\eta\iota} I'$, and $\vec{A} =_{\beta\eta\iota} \vec{A}'$.

Proof sketch A corollary of the confluence of TL (Theorem 3.3). □

Lemma 4.3 (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 [Wright and Felleisen 1994] using induction on the typing derivation for e. 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.

The interesting case is that of the dependently typed sel construct.

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

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

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 4.2 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

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

Also by Lemma 4.2 $A_2 =_{\beta\eta\iota} \widehat{n}$. 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 4.1(3)), the conclusion of $\mathcal{E}$ implies that $m < n$. Hence by rule R-SEL $e \mapsto v_m$.

The other cases are straightforward; as a representative, consider $e = e_1\ e_2$. If e_1 is not a value, then by inductive hypothesis $e_1 \mapsto e_1'$, therefore $e_1 = E_1\{e_{11}\}$ and $e_1' = E_1\{e_{11}'\}$ for some evaluation context E_1 and redex e_{11} such that $e_{11} \hookrightarrow e_{11}'$; then $e \mapsto E\{e_{11}'\}$, where $E = E_1\ e_2$. The subcase when e_1 is a value, but e_2 is not, is similar. If both e_1 and e_2 are values, then the typing derivation for e ends with an instance of rule E-CONV applied to a derivation with an instance of E-APP at its root, where a derivation for e_1 is in the premise for the subterm with an arrow type. Reasoning as in the case for sel above, since e_1 is a value and only rules E-FUN and E-FIX (again excluding E-CONV due to the standard form of the derivation) assign an arrow type to a value, we have that e_1 must be either an abstraction or a fixpoint (of an arrow type). Then e reduces by rule R-β or R-FIX, respectively, with the empty evaluation context. □

A standard type substitution lemma is used in the proof of Subject Reduction for the cases of redexes with type-level parameters.

Lemma 4.4 (Type substitution) If $\Delta, X{:}B; \Gamma \vdash\ e : A'$ and $\Delta \vdash\ A : B$, then $\Delta;\ [A/X]\Gamma \vdash\ [A/X]e : [A/X]A'$.

Proof sketch By induction on the typing derivation for e. □

Lemma 4.5 (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 use 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

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

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 4.4 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 $\cdot;\cdot \vdash\ [A'/X_1]e_1 : A$. □

Theorem 4.6 (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 4.3 and 4.5. □

4.6 Discussion

The proof of Progress of λ_H relies critically on the adequacy of the representation of meta-proofs of natural numbers being in the less-than relation, that is, that for closed A and B the kind $\mathsf{LT}\ A\ B$ is inhabited if *and only if* A and B represent natural numbers related by less-than. In the case of the less-than relation and LT this fact was proved in Lemma 4.1. However, it must be kept in mind when considering extensions of λ_H that since CIC and TL are more expressive than higher-order predicate logic, adequacy of the representations of meta-proofs does not hold in general, hence the existence of a term of the kind of the proposition does not imply that there is a meta-proof of the proposition. For instance the ability to eliminate inductive kinds in TL allows analysis of proof derivations—a technique which allows the construction of proof terms without counterpart in standard meta-reasoning. This issue does not arise for first-order proof representations (whose constructors have no parameters of a function kind) such as LT, 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 in place 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 [Appel and Felty 2000]); we plan to pursue this in the future.

The language λ_H is intended only as an illustration of the expressiveness of type systems based on TL. As we showed in Section 4.4, type conversions can be programmed in λ_H; however, it is also easy to extend λ_H with a type conversion construct cast, which allows conversion between any types which the programmer can prove are in a given relation of equivalence. The strongest such equivalence relation in TL is represented by Eq, and in this case the typing rule for cast is

$$\frac{\Delta;\ \Gamma \vdash\ e : A \qquad \Delta \vdash\ B : \mathsf{Eq}\ \Omega\ A\ A'}{\Delta;\ \Gamma \vdash\ \mathsf{cast}[A, A', B]e : A'} \qquad \text{(E-CAST)}$$

The dynamic semantics of cast is trivial:

$$\mathsf{cast}[A, A', B]e\ \hookrightarrow\ e \qquad \text{(R-CAST)}$$

The proof of the soundness of this extension is based on the observation (following from Theorem 3.3, the Church-Rosser property of TL) that if the judgment $\cdot\ \vdash\ B : \mathsf{Eq}\ \Omega\ A\ A'$ is derivable (which is what we have in the corresponding case of the proof of Subject Reduction), then the normal form B' of B is an application of refl to some kind equivalent to Ω and to some type A_1. But the kind of this application is then $\mathsf{Eq}\ \Omega\ A_1\ A_1$, while the kind of B' is $\mathsf{Eq}\ \Omega\ A\ A'$, so either $A = A_1$, or there is an application of rule CONV in the derivation for B', with a proof of $A =_{\beta\eta\iota} A_1$ in the premise, and similarly for A' vs. A_1. Thus we can obtain a proof that $A =_{\beta\eta\iota} A'$, and the rest of the meta-proof is the same as for E-CONV.[2]

In a language equipped with this construct, the programmer provides the compiler with proofs of correctness of type conversions, which legalizes more conversions than in any decidable type system with a built-in notion of conversion. Reusing definitions from Section 4.4, the cast from $\mathsf{snat}(\mathsf{plus}\ t\ t')$ to $\mathsf{snat}(\mathsf{plus}\ t'\ t)$ is

$$\begin{array}{l}\mathsf{cast}[\,\mathsf{snat}\ (\mathsf{plus}\ t\ t'),\\ \quad\ \mathsf{snat}\ (\mathsf{plus}\ t'\ t),\\ \quad\ \mathsf{eqf\ Nat}\ \Omega\ \mathsf{snat}\ (\mathsf{plus}\ t\ t')\ (\mathsf{plus}\ t'\ t)\ (\mathsf{plusSym}\ t\ t')]\\ \quad e\end{array}$$

[2]Again, this proof of soundness goes through with either an inductive definition of Eq, as in CIC, or with its Church encoding, since no large or dependent elimination of proof terms is used.

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. In this way, evaluation order is explicit and there is no need for a control stack.

There are two interesting points in our approach to CPS conversion. First, as we discuss in detail later in this section, arbitrary terms of the type language that appear in computation terms are not transformed. Second, the transformation of types is encoded as a function in our type language and, as will become apparent later in this section, this fact is important for proving that our CPS conversion is type-correct.

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.

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

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

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

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, fi x, which combines type and value abstraction. Multiple arguments may be passed by packing them in a tuple. We use the following syntactic sugar to denote non-recursive function definitions and value applications in λ_K(here x' is a fresh variable):

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

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:

$$\twoheadrightarrow \quad : \quad \Omega_K \rightarrow \Omega_K$$

We use the more conventional syntax $A \to \bot$ for $\twoheadrightarrow A$ (*i.e.,* the type of functions taking a parameter of type A). As will become apparent shortly in the static semantics of λ_K, no value of λ_K has type $A \to \bot$. The latter is used in conjunction with the new constructor func to form 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). In the case of function values, the type parameter of func is an element of Ω_K constructed by application of $\twoheadrightarrow$, $\forall_{\mathsf{Kind}}$ or $\forall_{\mathsf{Kscm}}$. The func constructor allows us to build one closure for each polymorphic function definition (even though it contains both type abstraction and term abstraction).

In the static semantics of λ_K we use two forms of judgments. As in λ_H, the judgment $\Delta; \Gamma \vdash_\kappa v : A$ indicates that the value v is well formed and of type A in the type and value contexts Δ and Γ respectively. Moreover, $\Delta; \Gamma \vdash_\kappa e$ 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 formation rules in Figure 6. We omit the rules for environment formation, variables, constants, tuples, packages, and type conversion on values, which are the same as in λ_H, and we give only one example for arithmetic and comparison operators. 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.

Typed CPS conversion involves the translation of both types and computation terms. Earlier algorithms [Harper and Lillibridge 1993; Morrisett et al. 1998] 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 previous algorithms, the translation also transforms the witness A:

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

Here we CPS-convert e and apply it to a continuation, which puts the result of its evaluation in a package and hands 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

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

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

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

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

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

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

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

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

Fig. 6. Static semantics of λ_K.

be defined as follows:

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

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

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

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

where

$$\mathsf{K_c} \equiv \lambda t{:}\Omega.\,\mathsf{func}\ (\mathsf{K}(t) \rightarrow \bot).$$

The definition of K is in the spirit of the interp function of Crary and Weirich [1999]. 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 its normal form B'. Clearly, $\mathsf{K}\ ([A/X]B) =_{\beta\eta\iota} \mathsf{K}\ ([A/X]B')$ and $[A/X](\mathsf{K}\ (B')) =_{\beta\eta\iota} [A/X](\mathsf{K}\ (B))$. Finally, we can show the equivalence $\mathsf{K}\ ([A/X]B') =_{\beta\eta\iota} [A/X](\mathsf{K}\ (B'))$ by induction over the structure of the normal form B'.

The definition of the CPS transformation for computation terms of λ_H to computation terms of λ_K is given in Figure 7. As an example of how CPS conversion works, let us consider the transformation of function abstraction $(\lambda x{:}A.\,e)$. The result is a function value that takes as a parameter a pair $x_{\arg}$, consisting of the original abstraction's parameter x and the current continuation k. After accessing the two elements of this pair, the function value applies the CPS conversion of the abstraction's body to k. On the other hand, the transformation of a function application $(e_1\ e_2)$ gives a function value that takes as a parameter the current continuation k. By applying the CPS conversions of e_1 and e_2 to appropriate continuations, this function value ultimately applies the function corresponding to e_1 to a pair consisting of the value corresponding to e_2 and the continuation k.

The following proposition states that our CPS conversion preserves typing. As we discussed earlier, it is important for its proof that K has been encoded as a function in TL.

Proposition 5.1 (Type Correctness of CPS Conversion)
If $\cdot;\cdot \vdash_H e : A$, then $\cdot;\cdot \vdash_K \mathcal{K}_{\mathsf{exp}}[\![\bar{e}^A]\!] : \mathsf{func}\ (\mathsf{K_c}(A) \rightarrow \bot)$.

Proof sketch By induction on the typing derivation for e. □

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.

Our approach to closure conversion is based on Morrisett *et al.* [Morrisett et al. 1998], who adopt a type-erasure interpretation of polymorphism. We use the same idea for existential types. As in the case of CPS conversion, there are again two interesting points in our approach. Arbitrary terms of the type language that appear in computation terms are not transformed. Moreover,

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

Fig. 7. CPS conversion: from λ_H to λ_K.

the transformation of types is again encoded as a function in our type language and this is crucial for proving that closure conversion is type-correct.

We call the language we use for this phase λ_C; its syntax is:

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

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.

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.

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

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

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

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 closure 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.

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

The definition of the closure transformation for the computation terms of λ_K is given in Figure 8. To understand how closure conversion works, let us again consider the transformations of function abstraction and function application. The former is the heart of closure conversion and clearly the most involved case. A λ_K term of the form $\mathsf{fi}\,\mathsf{x}\; x'[X_1{:}A_1, \ldots X_n{:}A_n](x{:}A).\, e$ is transformed to a package $\langle X = A_{\mathrm{env}}, \langle v_{\mathrm{code}}[Y_1] \ldots [Y_m], v_{\mathrm{env}} \rangle : A_X \rangle$. The first part of this package is the type of the closure environment A_{env}. The second part is a pair consisting of the transformed function body $v_{\mathrm{code}}[Y_1] \ldots [Y_m]$ and the closure environment v_{env}. The closure environment is a tuple containing the values of all term variables $x_0, \ldots x_{k-1}$ that are free in e. On the other hand, the transformed function body takes as parameters: (i) all type variables $Y_1, \ldots Y_m$ that are free in e, (ii) the type parameters $X_1, \ldots X_n$ of the original function, and (iii) a pair x_{arg} containing the closure environment x_{env} and the term parameter x of the original function. From the transformation of function abstractions, one immediately notices that quantification over kind schemas is required: the definition of ${A'}_X$ uses $\forall_{\mathsf{Kscm}}$ if $A_i : \mathsf{Kscm}$.

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

Fig. 8. Closure conversion: from λ_K to λ_C.

Inversely, the transformation of function application opens the package and reveals the type X_{env} and value x_{env} of the closure environment, as well as the function's body x_{code}. It then applies the body to the actual parameters and to x_{env}.

The following proposition states that our closure conversion preserves typing. As in the case of CPS conversion, the fact that Cl has been encoded as a function in TL is important for its proof.

Proposition 6.1 (Type Correctness of Closure Conversion) If $\cdot;\cdot \vdash_K v : A$, then $\cdot;\cdot \vdash_C \mathcal{C}_{\mathrm{val}}[\![v]\!] : \mathsf{Cl}\ (A)\ \bot$.

Proof sketch By induction on the typing derivation for v. □

7. RELATED WORK

Our type language is a variant of the calculus of constructions [Coquand and Huet 1988] extended with inductive definitions (with both small and large elimination) [Paulin-Mohring 1993; Werner 1994]. 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 Geuvers [1993] and Werner [1994] (which in turn borrows ideas from Altenkirch [1993]); 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 [Paulin-Mohring 1993]. 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.

Burstall and McKinna [1991] proposed the notion of deliverables, which is essentially the same as our notion of certified binaries. They use dependent strong sums 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 [Xi and Pfenning 1999] 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. DML does not support explicit proofs in its type language; any assertions (or constraints) must be resolved fully automatically in order to ensure decidable typechecking. As a result, DML's assertion language only allows integer linear inequalities. Our system, on the other hand, allows arbitrary propositions and proofs. An assertion in our system can use any integer constraints but a certified program must explicitly provide proofs on how these constraints are satisfied. Our system is best suited for use in compiler typed intermediate languages while the DML type system is more suitable for use in a source programming language. Another difference is that DML does not define the Ω kind as an inductive definition so it does not support intensional type analysis [Trifonov et al. 2000] and it is unclear how it can preserve 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 [Harper and Morrisett 1995; Crary and Weirich 1999; Trifonov et al. 2000]; the type-analysis construct in the computation language can be eliminated using the technique proposed by Crary et al. [1998].

The work presented in this paper showed one way of having types and proofs coexist in an intermediate language for certified binaries, that is, by embedding predicates and proofs directly into types. Another possibility, which we did not address, is to embed types into the logic which proofs are carried out—

essentially using pre- and post-conditions as in Hoare logic to express type invariants. Unfortunately, Hoare logic does not work well with higher-order functions, for example, it is unclear how to describe an assertion that a formal parameter (of another function) has a function type (as simple as int→int). Foundational PCC [Appel and Felty 2000] requires explicit construction of the fixed point (using index-based semantic model) to support higher-order functions—which is probably too complex for compiler intermediate languages.

Concurrently with our work, Crary and Vanderwaart [2001] recently proposed a system called LTT, which also aims at adding explicit proofs to typed intermediate languages. LTT uses Linear LF [Cervesato and Pfenning 1996] as its proof language. It shares some similarities with our system in that both are using singleton types [Xi and Pfenning 1999] to circumvent the problems of dependent types. However, since LF does not have inductive definitions and the Elim construct, it is unclear how LTT can support intensional type analysis and type-level primitive recursive functions [Crary and Weirich 2000]. In fact, to define Ω as an inductive kind [Trifonov et al. 2000], 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 definition. We hope to evolve our framework into a realistic typed common intermediate format.

APPENDIX

In this appendix we supply the rest of the details involved in the formalization of our type language TL.

A. FORMALIZATION OF TL

Most of our notation and definitions are directly borrowed from Werner [1994]. 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.

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 Definitions A.1 and A.2.

Definition A.1 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 A.2 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) $B' \rightarrow C'$, where B' 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 when C is of the form $B \rightarrow C'$ and X does not occur free in B. Therefore, we only allow the occurrence of X in the non-dependent case.

In the rest of this paper we often write well-formed constructor kinds 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 A.3 Let C be a well-formed constructor kind for X. Then C is of the form $\Pi\vec{Y}:\vec{B}.\,X$. If all the Y's are t's, that is, C is of the form $\Pi\vec{t}:\vec{B}.\,X$, then we say that C is a small constructor kind (or just a 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. 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 A.4 Let C be a well-formed constructor kind for X and let A, B', and I be terms. We define $\Phi_{X,I,B'}(C,A)$ inductively on the structure of C:

$$\begin{array}{lcl}
\Phi_{X,I,B'}(X, A) & \stackrel{\text{def}}{=} & A \\
\Phi_{X,I,B'}(\Pi Y : B.\, C', A) & \stackrel{\text{def}}{=} & \lambda Y : B.\, \Phi_{X,I,B'}(C', A\ Y) \\
\Phi_{X,I,B'}((\Pi \vec{Y} : \vec{B}.\, X) \to C', A) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\lambda Z : (\Pi \vec{Y} : \vec{B}.\, I).\, \Phi_{X,I,B'}(C', A\ Z\ (\lambda \vec{Y} : \vec{B}.\, B'\ (Z\ \vec{Y})))}
\end{array}$$

Definition A.5 The reduction relations on our terms are defined as:

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

Recall that in Section 3.2 we introduced the relations $\rhd_\beta$, $\rhd_\eta$, and $\rhd_\iota$ as the contextual closures of the relations $\leadsto_\beta$, $\leadsto_\eta$, and $\leadsto_\iota$ respectively; we write $\leadsto$ and $\rhd$ for the unions of the above relations, and $=_{\beta\eta\iota}$ for the reflexive, symmetric, and 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 of 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:

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

In the case when $C_i = \Pi \vec{Y} : \vec{B}.\, X$, where X does not occur free in $\vec{B}$, A must be of the form $\mathsf{Ctor}\,(i, I)\ \vec{A}$, with A_i of type B_i. The Elim selects the B_i branch and passes the constructor arguments to it. Accordingly, the reduction yields (by application of the meta-level function Φ):

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

The recursive case is the most interesting. For simplicity assume that the ith 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 $\Pi \vec{Y} : \vec{B'}.\, I$, and $A_2 \ldots A_n$ being non-recursive. The reduction rule then yields:

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

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.

For example, suppose the kind Nat of natural numbers is defined as

$$\mathsf{Ind}(\mathsf{Nat} : \mathsf{Kind})\{\mathsf{Nat}; \mathsf{Nat} \to \mathsf{Nat}\},$$

with the constructor zero defined as $\mathsf{Ctor}\,(1, \mathsf{Nat})$ and the constructor succ defined as $\mathsf{Ctor}\,(2, \mathsf{Nat})$. Consider $\mathsf{Elim}[\mathsf{Nat}, A''](A)\{B_0; B_S\}$, where B_0 and B_S are the branches for the zero and succ constructors. Then we have:

$$\begin{array}{l}
\mathsf{Elim}[\mathsf{Nat}, A''](\mathsf{Ctor}\,(1, \mathsf{Nat}))\{B_0; B_S\} \leadsto_\iota B_0 \\
\mathsf{Elim}[\mathsf{Nat}, A''](\mathsf{Ctor}\,(2, \mathsf{Nat})\ N)\{B_0; B_S\} \leadsto_\iota B_S\ N\ (\mathsf{Elim}[\mathsf{Nat}, A''](N)\{B_0; B_S\})
\end{array}$$

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Delta \vdash A : I \qquad \Delta \vdash A' : \mathsf{Kscm} \\ \mathit{for\ all}\ i \qquad \mathit{small}(C_i) \qquad \Delta \vdash B_i : \Psi_{X,I}(C_i, A')\end{array}}{\begin{array}{c}\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A' \\ \text{where } I = \mathsf{Ind}(X{:}\mathsf{Kind})\{\vec{C}\} \\ \Delta \text{ binds no kind-schema variables}\end{array}} \qquad \text{(L-ELIM)}$$

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

Fig. 9. Formation rules of TL.

The following two definitions introduce the meta-level functions ζ and Ψ, which compute the types of the branches of the small and large elimination constructs, respectively. The cases follow from the ι-reduction rule in Definition A.5.

Definition A.6 Let C be a well-formed constructor kind for X and let A, B', and I be terms. We define $\zeta_{X,I}(C, A, B')$ inductively on the structure of C:

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

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

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

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

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

The complete typing rules for TL are listed in Figure 9. The three weakening rules make sure that all variables are bound to the correct 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).

ACKNOWLEDGMENTS

We would like to thank Thorsten Altenkirch, Gilles Barthe, Thierry Coquand, Antony Courtney, Karl Crary, Xinyu Feng, Christopher League, Zhaohui Luo, Christine Paulin-Mohring, Stefan Monnier, Henrik Nilsson, Walid Taha, and anonymous referees for discussions and comments on this and an earlier version of this paper. Benjamin Werner helped us understand the intricacies in the strong-normalization proof for the core calculus of inductive constructions.

REFERENCES

ALTENKIRCH, T. 1993. Constructions, inductive types and strong normalization. Ph.D. thesis, University of Edinburgh, UK.

APPEL, A. W. AND FELTEN, E. W. 2001. Models for security policies in proof-carrying code. Tech. Rep. CS-TR-636-01, Princeton Univ., Dept. of Computer Science. March.

APPEL, A. W. AND FELTY, A. P. 2000. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.* ACM Press, 243–253.

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

BARENDREGT, H. P. AND GEUVERS, H. 1999. Proof-assistants using dependent type systems. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Elsevier Sci. Pub. B.V.

BARTHE, G., HATCLIFF, J., AND SORENSEN, M. 1999. CPS translations and applications: the cube and beyond. *Higher Order and Symbolic Computation 12,* 2 (September), 125–170.

BURSTALL, R. AND MCKINNA, J. 1991. Deliverables: an approach to program development in constructions. Tech. Rep. ECS-LFCS-91-133, Univ. of Edinburgh, UK.

CERVESATO, I. AND PFENNING, F. 1996. A linear logical framework. In *Proc. 11th IEEE Symp. on Logic in Computer Science*. 264–275.

COLBY, C., LEE, P., NECULA, G. C., BLAU, F., PLESKO, M., AND CLINE, K. 2000. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.* ACM Press, New York, 95–107.

CONSTABLE, R. 1985. Constructive mathematics as a programming logic I: Some principles of theory. *Ann. of Discrete Mathematics 24*.

COQUAND, T. AND HUET, G. 1988. The calculus of constructions. *Information and Computation 76*, 95–120.

CRARY, K. AND VANDERWAART, J. 2001. An expressive, scalable type theory for certified code. Tech. Rep. CMU-CS-01-113, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA. May.

CRARY, K., WALKER, D., AND MORRISETT, G. 1999. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.* ACM Press, 262–275.

CRARY, K. AND WEIRICH, S. 1999. Flexible type analysis. In *Proc. 1999 ACM SIGPLAN Int'l Conf. on Functional Prog.* ACM Press, 233–248.

CRARY, K. AND WEIRICH, S. 2000. Resource bound certification. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.* ACM Press, 184–198.

CRARY, K., WEIRICH, S., AND MORRISETT, G. 1998. Intensional polymorphism in type-erasure semantics. In *Proc. 1998 ACM SIGPLAN Int'l Conf. on Functional Prog.* ACM Press, 301–312.

GEUVERS, H. 1993. Logics and type systems. Ph.D. thesis, Catholic University of Nijmegen, The Netherlands.

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

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

HARPER, R. AND LILLIBRIDGE, M. 1993. Explicit polymorphism and CPS conversion. In *Proc. 20th ACM Symp. on Principles of Prog. Lang.* ACM Press, 206–219.

HARPER, R. AND MORRISETT, G. 1995. Compiling polymorphism using intensional type analysis. In *Proc. 22nd ACM Symp. on Principles of Prog. Lang.* ACM Press, 130–141.

HAYASHI, S. 1991. Singleton, union and intersection types for program extraction. In *Proc. International Conference on Theoretical Aspects of Computer Software*, A. R. Meyer, Ed. 701–730.

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

HUET, G., PAULIN-MOHRING, C., ET AL. 2000. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *Proc. 23rd ACM Symp. on Principles of Prog. Lang.* ACM Press, 271–283.

MONNIER, S., SAHA, B., AND SHAO, Z. 2001. Principled scavenging. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.* ACM Press, New York, 81–91.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.* ACM Press, 85–97.

NECULA, G. 1997. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.* ACM Press, New York, 106–119.

NECULA, G. 1998. Compiling with proofs. Ph.D. thesis, School of Computer Science, Carnegie Mellon Univ.

NECULA, G. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.* USENIX Assoc., 229–243.

NECULA, G. AND LEE, P. 1998. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.* ACM Press, New York, 333–344.

NORDSTROM, B., PETERSSON, K., AND SMITH, J. 1990. *Programming in Martin-Löf's type theory*. Oxford University Press.

PAULIN-MOHRING, C. 1989. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Proc. 16th ACM Symp. on Principles of Prog. Lang.* ACM Press, New York, 89–104.

PAULIN-MOHRING, C. 1993. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, M. Bezem and J. Groote, Eds. LNCS 664, Springer-Verlag.

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

SHAO, Z., LEAGUE, C., AND MONNIER, S. 1998. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN Int'l Conf. on Functional Prog.* ACM Press, 313–323.

SHAO, Z., SAHA, B., TRIFONOV, V., AND PAPASPYROU, N. 2001. A type system for certified binaries. Tech. Rep. YALEU/DCS/TR-1211, Dept. of Computer Science, Yale University, New Haven, CT. March.

SHELDON, M. A. AND GIFFORD, D. K. 1990. Static dependent types for first class modules. In *1990 ACM Conference on Lisp and Functional Programming*. ACM Press, New York, 20–29.

TRIFONOV, V., SAHA, B., AND SHAO, Z. 2000. Fully reflexive intensional type analysis. In *Proc. 2000 ACM SIGPLAN Int'l Conf. on Functional Prog.* ACM Press, 82–93.

WALKER, D. 2000. A type system for expressive security policies. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.* 254–267.

WERNER, B. 1994. Une théorie des constructions inductives. Ph.D. thesis, A L'Université Paris 7, Paris, France.

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

XI, H. AND PFENNING, F. 1999. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.* ACM Press, 214–227.

Received January 2002; revised March 2003 and November 2003; accepted May 2004.