A Type System for Certified Binaries
Zhong Shao  Bratin Saha  Valery Trifonov  Nikolaos Papaspyrou
Yale University
YALETU/DCS/TR-1211
July 23, 2001

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

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

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

Technical Report YALEU/DCS/TR-1211
March 2001 (revised July 2001)

Abstract

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

1 Introduction

Proof-carrying code (PCC), as pioneered by Necula and Lee [30, 29], 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 [32, 1]. For example, the Foundational PCC system [2] can certify any property expressible in Church’s higher-order logic. Harper et al. [19, 6] call all these proof-carrying constructs certified binaries (or deliverables [6]). A certified binary is a value (which can be a function, a data structure, or a combination of both) together with a proof that the value satisfies a given specification.

Unfortunately, little is known on how to construct or generate certified binaries. Existing certifying compilers [31, 8] have focused on simple memory and control-flow safety only. Typed intermediate languages [22] and typed assembly languages [28] are effective techniques for automatically generating certified code; however, none of these type systems can rival the expressiveness of the actual higher-order logic as used in some PCC systems [2].

In this paper, we present a type-theoretic framework for constructing, composing, and reasoning about certified binaries. Our plan is to use the formulae-as-types principle [24] to represent propositions and proofs in a general type system, and then to investigate their relationship with compiler intermediate and assembling languages. We show how to integrate an entire proof system (the calculus of inductive constructions [35, 10]) into an intermediate language, and how to define complex transformations (CPS and closure conversion) of programs in this language so that they preserve proofs represented in the type system. Our paper builds upon a large body of previous work in the logic and theorem-proving community (see Barendregt et al. [4, 3] for a good summary), and makes the following new contributions:

- We show how to design new typed intermediate languages that are capable of representing and manipulating propositions and proofs. In particular, we show how to maintain decidability of typechecking when reasoning about certified programs that involve effects. This is different from the work done in the logic community which focuses on strongly normalizing (primitive recursive) programs.
- We maintain a phase distinction between compile-time type-checking and run-time evaluation. This property is often lost in the presence of dependent types (which are necessary for representing proofs in predicate logic). We achieve this by never having the type language (see Section 3) dependent on the computation language (see Section 4). Proofs are instead always represented at the type level using dependent kinds.
- We show how to use propositions to express program invariants and how to use proofs to serve as static capabilities. Following Xi and Pfenning [44], we use singleton types [23] to support the necessary interaction between the type and computation languages. We can assign an accurate type to unchecked vector (or array) access (see Section 4.2). Xi and Pfenning [44] can achieve the same using constraint checking, but their system does not support arbitrary propositions and (explicit) proofs, so it is less general than ours.
- We use a single type language to typecheck different compiler intermediate languages. This is crucial because it is impractical to have separate proof libraries for each intermediate language. We achieve this by using inductive definitions to define all types used to classify computation terms. This in turn nicely fits our work on (fully reflexive) intensional type analysis [39] into a single system.
- We show how to perform CPS and closure conversion on our intermediate languages while still preserving proofs represented in the type system. Existing algorithms [28, 21, 26, 5] all require that the transformation be performed on the entire type language. This is impractical because proofs are large in size; transforming them can alter their meanings and break the sharing among different languages. We present new techniques that completely solve these problems (Sections 5–6).

This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.
Our type language is a variant of the calculus of inductive constructions [35, 10]. Following Werner [41], we give rigorous proofs for its meta-theoretic properties (subject reduction, strong normalization, confluence, and consistency of the underlying logic). We also give the soundness proof for our sample computation language. See Sections 3 and 4, and the appendix for more 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 [22, 44, 12, 14, 40, 39] has produced many specialized, increasingly complex type systems, each with its own meta-theoretical proofs, yet it is unclear how they will fit together. We can hope to replace them with one very general type system whose meta theory is proved once and for all, and that allows the definition of specialized type operators via the general mechanism of inductive definitions. For example, inductive definitions subsume and generalize earlier systems on intensional type analysis [22, 13, 39].

We have started implementing our new type system in the FLINT compiler [36, 37], but making the implementation realistic still involves solving many remaining problems (e.g., efficient proof implementations). 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 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. 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.

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

We divide each typed intermediate language into a type sublanguage 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 $τ_1 → τ_2$ and the polymorphic type $∀τ : κ. τ$ 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 $σ$” to imply that $σ$ has kind $κ → · · · → Ω$.” Kind terms with other kind schemas, or type terms with other kinds are strictly referred as “kind terms” or “type terms.”

As far as we know, our work is the first comprehensive study on how to incorporate higher-order predicate logic (with inductive

### THE TYPE LANGUAGE:

(kscm) $u ::= \text{Kind} | \ldots$

(kind) $κ ::= κ_1 → κ_2 | Ω | \ldots$

(type) $τ ::= t | λt : κ. τ | τ_1 τ_2 | ∀t : κ. τ | \ldots$

### THE COMPUTATION LANGUAGE:

(exp) $e ::= x | λx : τ. e | e_1 e_2 | Λ : κ. e | e[τ] | \ldots$

Figure 1: Typed λ-calculi—a skeleton

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

2.1 Representing propositions and proofs

The first step is to represent propositions and proofs for a particular logic in a type-theoretic setting. The most established technique is to use the $\text{formulas-as-types}$ principle (a.k.a. the Curry-Howard correspondence) [24] to map propositions and proofs into a typed λ-calculus. The essential idea, which is inspired by constructive logic, is to use types (of kind $Ω$) to represent propositions, and expressions to represent proofs. A proof of an implication $P → Q$ is a function object that yields a proof of proposition $Q$ when applied to a proof of proposition $P$. A proof of a conjunction $P ∧ Q$ 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 $P ∨ Q$ is a pair $(b, e)$—a tagged union—where $b$ is either 0 or 1 and if $b = 0$, $e$ 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 $∀x ∈ B. P(x)$ 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 $∃x ∈ B. P(x)$ 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 type-checking in the corresponding typed λ-calculus. There has been a large body of work done along this line in the last 30 years; most type-based proof assistants are based on this fundamental principle. Barendregt et al. [4, 3] give a good survey on previous work in this area.

2.2 Representing certified binaries

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

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

Here $e$ is just an expression with type $P(v)$. The predicate $P$ is a dependent type constructor with kind $τ → Ω$. The entire package $S$ has a dependent strong-sum type $Σx : τ. P(x)$.

For example, suppose $\text{Nat}$ is the domain for natural numbers and $\text{Prime}$ is a unary predicate that asserts an element of $\text{Nat}$ as a prime number, we introduce a type nat representing $\text{Nat}$, and a type constructor prime (of kind $\text{nat} → Ω$) representing $\text{Prime}$. We can build a certified prime-number package by pairing a value $v$
(a natural number) with a proof for the proposition prime(v); the resulting certified binary has type \( \Sigma x : \text{nat.} \; \text{prime}(x) \).

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:

\[
\forall x \in \text{Nat.} \; \text{Prime}(x) \supset \text{Prime}(f(x))
\]

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

\[\Pi x : \text{nat.} \; \text{prime}(x) \rightarrow \text{prime}(f(x))\]

The resulting certified binary has type

\[\Sigma f : \text{nat} \rightarrow \text{nat.} \; \Pi x : \text{nat.} \; \text{prime}(x) \rightarrow \text{prime}(f(x))\]

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

### 2.3 The problems with dependent types

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

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

Even if applying \( f \) does not involve any effects, we still have more serious problems. In a type-preserving compiler, the body of the function \( f \) has to be compiled down to typed low-level languages. A few compilers perform typed CPS conversion [28], but in the presence of dependent types, this is still an open problem [5]. Also, typechecking in low-level languages would now require performing the equivalent of \( \beta \)-reductions on the low-level (assembly) code; this is awkward and difficult to support cleanly.

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

Finally, it would be nice to support a notion of subset types [9, 33]. A certified binary of type \( \Sigma x : \text{nat.} \; \text{prime}(x) \) contains a natural number \( v \) and a proof that \( v \) is a prime. However, in some cases, we just want \( v \) to belong to a subset type \( \{ x : \text{nat} \mid \text{prime}(x) \} \), i.e., \( v \) is a prime number but the proof of this is not together with \( v \); instead, it can be constructed from the current context.

### 2.4 Separating the type and computation languages

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

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

To maintain decidable typechecking, we insist that the type language no longer depends on any runtime computation. Given a type-level function \( g \) of kind \( \text{Nat} \rightarrow \text{Nat} \), we can certify that it always maps a prime to another prime by building a proof \( \tau_p \). For the following proposition, now represented as a dependent product kind:

\[\Pi t : \text{Nat.} \; \text{Prime}(t) \rightarrow \text{Prime}(g(t))\]

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

To reason about actual programs, we still have to connect terms in the type language with those in the computation language. We follow Xi and Pfenning [44] and use singleton types [23] to relate computation values to type terms. In the previous example, we introduced a singleton type constructor \( \text{snat} \) of kind \( \text{Nat} \rightarrow \Omega \). Given a type term \( n \) of kind \( \text{Nat} \), if a computation value \( v \) has type \( \text{snat}(n) \), 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 \( \text{Nat} \), a proof for the proposition \( \text{Prime}(n) \), and a computation value of type \( \text{snat}(n) \). We can pack it up into an existential package and make it a first-class value with type:

\[\exists n : \text{Nat.} \; \exists t : \text{Prime}(n) \cdot \text{snat}(n)\]

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

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

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

\[
\forall n : \text{Nat.} \; \text{snat}(n) \rightarrow \text{snat}(g(n))
\]

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 \( \text{snat}(n) \) as input and does not loop forever, then it will return an integer of type \( \text{snat}(g(n)) \).

Second, we construct a proof \( \tau_p \) 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 \( \tau_p \), and the computation function \( f \) itself. We can pack it into an existential package with type:

\[\exists : \text{Nat} \rightarrow \text{Nat.} \; \exists p : (\Pi t : \text{Nat.} \; \text{Prime}(t) \rightarrow \text{Prime}(g(t))) \cdot \forall n : \text{Nat.} \; \text{snat}(n) \rightarrow \text{snat}(g(n))\]

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.
We can also restrict \( f \) so that it can only be applied to prime numbers; all we need is to add an additional proof argument, so \( f \) has type:

\[
\forall n : \text{Nat}. \forall t : \text{Prime}(n). \text{snat}(n) \rightarrow \text{snat}(g(n)).
\]

Here, the parameter \( t \) serves as a static capability; a proof for \( \text{Prime}(n) \) exists only if \( n \) is indeed a prime.

### 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 \( \Omega \)) to classify the computation terms. A typical compiler intermediate language supports a large number of base 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 \( \Omega \); for example, a computation function at the source level may be turned into CPS-style, or later, to one whose arguments are machine registers [28]. We also want to support intentional type analysis [22] which is crucial for type-checking runtime services [27].

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

\[
\text{(kscm)} \quad u ::= \text{Kind} \mid \ldots
\]

\[
\text{(kind)} \quad \kappa ::= \kappa_1 \rightarrow \kappa_2 \mid k \mid \Pi k : u. \kappa \mid \ldots
\]

\[
\text{(type)} \quad \sigma ::= t \mid \lambda \kappa : \kappa. \sigma \mid \tau_1 \tau_2 \mid \lambda k : u. \sigma \mid \sigma[k] \mid \ldots
\]

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

\[
\text{Inductive } \Omega : \text{Kind} ::= \rightarrow : \Omega \rightarrow \Omega \rightarrow \Omega \quad | \quad \forall : \Pi k : \text{Kind}. (k \rightarrow \Omega) \rightarrow \Omega
\]

Here \( \rightarrow \) and \( \forall \) are two of the constructors (of \( \Omega \)). The polymorphic type \( \forall \kappa. \tau \) is now written as \( \forall \kappa \) (\( \lambda \kappa : \kappa. \tau \)); the function type \( \tau_1 \rightarrow \tau_2 \) is just \( \tau_1 \tau_2 \).

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 conditional kind \( \Pi k : \kappa_1, k_2 \), which subsumes the arrow kind \( \kappa_1 \rightarrow k_2 \); we also add kind-level functions to represent predicates. Thus the type language naturally becomes the calculus of inductive constructions [35].

Notice standard formulation of Church’s higher-order logic puts propositions at the same level as terms (which are type terms in our setup); proofs are then represented at a level below (parallel to our computation language). This does not work since we already require polymorphic kinds for the inductive definition of \( \Omega \); with impredicative polymorphism on both the kind and type levels, the proof language becomes Girard’s \( \lambda \text{U} \) [18] which is known to be inconsistent.

### 2.6 Proof-preserving compilation

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

These challenges are non-trivial but the way we set up our type system makes it easier to solve them. First, because our type language does not depend on the computation language, we do not have the difficulties involved in CPS-converting dependently typed \( \lambda \)-calculi [5]. Second, all our intermediate languages share the same type language thus also the same proof library; this is possible because the \( \Omega \) 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 \( \Omega \) kind) into the target types (of the target \( \Omega \) kind). Existing CPS- and closure-conversion algorithms [28, 21, 26] all perform such translation at the meta-level; they have to go through every type term (thus every proof term in our setting) during the translation, because any type term may contain a sub-term which has the source \( \Omega \) kind. In our framework, the fact that each \( \Omega \) 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 \( \Omega \) kind. In the rest of this paper, we first give a formal definition of our type language (which will be named as TL from now on) in Section 3; we then present a sample computation language \( \lambda_H \) in Section 4; we show how \( \lambda_H \) 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 [25]. 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 [34] which are not required in our case (where proofs are only used as specifications for the computation terms). Second, as far as we know, there are still no formal studies covering the entire CIC language. Third, for theoretical purposes, we want to understand what are the most essential features for modeling certified binaries.

**Motivations** Following the discussion in Section 2.5, we organize TL into the following three levels:
A good way to comprehend TL is to look at its five 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., \(\lambda k.\) kind variables \(\kappa,\) kind variables \(u,\) kind variables \(t\)) to 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 \(t.\)

TL has an additional level above \(\text{Kscm},\) of which \(\text{Kscm}\) is the sole member.

### Kind schemas \((\text{Kscm})\) classify kind terms while kinds classify type terms.

- **Kinds** \(\Pi: \kappa_1, \kappa_2\) and \(\kappa_1 \to \kappa_2\) are used to typecheck the type-level function \(\lambda \tau. \kappa, \tau\) and its application form \(\tau_1 \tau_2.\)
- **Prime** is a predicate with kind schema Nat \(\to\) Kind, we can write a type term such as \(\lambda \Omega. \Omega \to \Omega,\) a type-level arithmetic function such as plus which has kind \(\text{Nat} \to \text{Nat} \to \text{Nat},\) or the universally quantified proposition in Section 2.2 which is represented as a kind \(\Pi: \kappa_1, \kappa_2\) (e.g., \(\Pi: \kappa_1, \kappa_2\) is non-dependent if \(t\) does not occur free in \(\kappa_2).\)

### Kinds \((\Pi)\) to Nat.

- **Kinds** \(\Pi: \kappa, u, \kappa\) and \(u \to \kappa\) are used to typecheck the type-level kind abstraction \(\lambda \tau. \kappa, \tau\) and its application form \(\tau[u].\)
- **Inductive kinds** \(\Pi: \kappa_1, \kappa_2\) and \(\exists \kappa_1, \kappa_2\) have constructors whose kinds are specified by \(\kappa.\)

**Figure 2:** Examples of inductive definitions

- **Inductive** \(\text{Bool}\) : Kind := true : Bool | false : Bool

- **Inductive** \(\text{Nat}\) : Kind := zero : Nat | succ : Nat \to Nat

### Inductive kinds \((\Pi)\) to \(\text{Nat}\) and \(\text{Bool}\).

- **Kinds** \(\Pi: u_1, u_2\) and \(u_1 \to u_2\) are used to typecheck the kind-level function \(\lambda \kappa. u, \kappa\) and its application form \(u_1 \to u_2\). We use it to write higher-order predicates and logical connectives. For example, the logical negation operator can be written as follows:

\[
\text{Not} : \text{Kind} \to \text{Kind} = \lambda k. \text{Kind}. (k \to \text{False})
\]

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

TL also provides a general mechanism of inductive definitions [35]. The term \(\text{Ind}(k: \text{Kind})\) introduces an inductive kind \(k\) containing a list of constructors whose kinds are specified by \(\vec{k}.\) Here \(k\) must only occur “positively” inside each \(\vec{k}.\) (see Appendix D for the formal definition of positivity). The term \(\text{Ctor}(i, \vec{k})\) refers to the \(i\)-th constructor in an inductive kind \(k.\) For presentation, we will use a more friendly syntax in the rest of this paper. An inductive kind \(I = \text{Ind}(k: \text{Kind})\) will be written as:

\[
\text{Inductive} I : \text{Kind} := \cdots
\]

We give an explicit name \(c_i\) to each constructor, so \(c_i\) is just an abbreviation of \(\text{Ctor}(i, I).\) For simplicity, the current version of TL does not include parameterized inductive kinds, but supporting them is quite straightforward [41, 35].

TL provides two iterators to support primitive recursion on inductive kinds. The small elimination \(\text{Elim}(\kappa, u)(\vec{k})\) takes a type term \(\tau\) of inductive kind \(\kappa\), performs the iterative operation specified by \(\vec{\tau}\) (which contains a branch for each constructor of \(\vec{k}\)), and returns a type term of kind \(\kappa.\tau\) as the result. The large elimination \(\text{Elim}(\kappa, u)(\vec{k})\) takes a type term \(\tau\) of inductive kind \(\kappa\), performs the iterative operation specified by \(\vec{\tau}\), and returns a kind

\[
\text{LT} : \text{Nat} \to \text{Nat} \to \text{Kind}
\]

so that \(\text{LT} t_1 t_2\) is a proposition asserting that the natural number represented by \(t_1\) is less than that of \(t_2.\)
term of kind schema \( u \) as the result. These iterators generalize the Typerec operator used in intensional type analysis [22, 13, 39].

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

\[
\begin{align*}
\text{Elim}[\text{Nat}, \kappa](\text{zero}) & \{ \tau_1; \tau_2 \} \leadsto \tau_1 \\
\text{Elim}[\text{Nat}, \kappa](\text{succ } \tau) & \{ \tau_1; \tau_2 \} \leadsto \tau_2 \ (\text{Elim}[\text{Nat}, \kappa](\tau') \{ \tau_1; \tau_2 \})
\end{align*}
\]

The general \( \iota \)-reduction rule is defined formally in Appendix D. 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 \( \iota \) evaluates to true if its first argument is less than or equal to the second. The function \( \eta \) performs the less-than comparison.

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

Formalization. We want to give a formal semantics to TL and then reason about its meta-theoretical properties. But the five II constructs have many redundancies, so in the rest of this paper, we will model TL as a pure type system (PTS) [3] extended with inductive definitions. Intuitively, instead of having a separate syntactic category for each level, we collapse all kind schemas \( u \), kind terms \( \kappa \), type terms \( \tau \), and the external constant \( \text{Kscm} \) into a single set of pseudo-term s (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 II construct (\( \Pi \times : A : B \)), one \( \lambda \)-abstraction (\( \lambda \times : A : B \)), and one application form (\( A : B \)); two inductive definitions are also merged into one \( \text{Elim}[A', B'](\{ A \} \{ B \}) \). We use \( X \) and \( Y \) to represent generic variables, but we will still use \( t, k, \) and \( z \) if the class of a variable is clear from the context.

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

\[
\begin{align*}
\mathcal{S} &= \text{Kind, Kscm, Ext} \\
\mathcal{A} &= \text{Kind : Kscm, Kscm : Ext} \\
\mathcal{R} &= (\text{Kind, Kind}), (\text{Kscm, Kind}), (\text{Ext, Kind}) \\
&\quad (\text{Kind, Kscm}), (\text{Kscm, Kscm})
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Systems</th>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>TL</td>
<td>Kind</td>
</tr>
<tr>
<td>Werner [41]</td>
<td>Kscm</td>
</tr>
<tr>
<td>Coq/Cic [25]</td>
<td>Ext</td>
</tr>
<tr>
<td>Barendregt [3]</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{PTs rules/ptm} & \mathcal{P} & \lambda \times : A : B & A \\
\hline
\langle \text{Kind, Kind} \rangle & \Pi \kappa_1 \times \kappa_2 & \lambda \kappa \times \tau & \tau_1 \tau_2 \\
\langle \text{Kscm, Kind} \rangle & \Pi \lambda \kappa \times \tau & \lambda \kappa \times \tau & \tau \kappa \\
\langle \text{Ext, Kind} \rangle & \Pi \mathcal{T} \kappa \times \tau & \lambda \kappa \times \tau & \tau \kappa \\
\langle \text{Kind, Kscm} \rangle & \Pi \kappa_1 \times \kappa_2 & \lambda \kappa \times \kappa & \kappa' \\
\langle \text{Kscm, Kscm} \rangle & \Pi \kappa_1 \times \kappa_2 & \lambda \kappa \times \kappa & \kappa' \\
\hline
\end{array}
\]

We define a context \( \Delta \) as a list of bindings from variables to pseudoterms:

\[
\Delta ::= \cdot | \Delta, X : A
\]

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

\[
\Delta \vdash \langle A \rangle : \mathcal{P} \vdash \langle B \rangle : A : A' \quad \vdash \Pi \times : A : B : s_1 \quad \vdash \Pi \times : A : B : s_2 \quad \text{s}_1 \in \mathcal{R}
\]

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

\[
\Delta, \lambda : \mathcal{P} : A \vdash B : B' \quad \Delta \vdash \Pi \times : A : B' : s
\]

The reduction rules can also be shared. TL supports the standard \( \beta \) and \( \eta \)-reductions (denoted as \( \sim_\beta \text{ and } \sim_\eta \)) plus the previously mentioned \( \iota \)-reductions (denoted as \( \sim_\iota \)) on inductive objects (see Appendix D). We use \( \iota_\beta \text{, } \iota_\eta \text{, and } \iota_\lambda \) to denote the relations that correspond to the rewriting of subterms using the relations \( \sim_\beta \text{, } \sim_\eta \text{, and } \sim_\lambda \) respectively. We use \( \sim_\iota \) for the unions of the above relations. We also write \( \iota_\beta \) for the reflexive-symmetric-transitive closure of \( \iota \).

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

4 The Computation Language \( \lambda_H \)

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

One approach to presenting a language of computations is to encode its syntax and semantics in a proof system, with the benefit of obtaining machine-checkable proofs of its properties, e.g. type safety. This appears to be even more promising for a system with a type safety like CIC, which is more expressive than higher-order predicate logic: The CIC proofs of some program properties, embedded as type terms in the program, may not be easily representable in meta-logical terms, thus it may be simpler to perform all the reasoning in CIC. However our exposition of the language TL is focused on its use as a type language, and consequently it does not include all features of CIC. We therefore leave this possibility for future work, and give a standard meta-logical presentation instead; we address some of the issues related to adequacy in our discussion of type safety.

In this section we often use the unqualified “term” to refer to a computation term (expression) e, with syntax defined in Figure 4. Most of the constructs are borrowed from standard higher-order typed calculi. To simplify the exposition we only consider constants representing natural numbers (π 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 term. Dually, the type argument in a type application, and the witness type term A in the package construction \( \langle X = A, e : A' \rangle \) can be either a type term or a kind term.

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

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

\[
v ::= \pi | \text{tt} | \text{ff} | f | \text{fix} \ f : A . \ f | \langle X = A, v : A' \rangle | \langle v_0, \ldots, v_{n-1} \rangle
\]

The reduction relation \( \rightsquigarrow \) is specified by the rules

\[
\begin{align*}
\text{(exp)} & \quad e ::= x | \pi | \text{tt} | \text{ff} | f | \text{fix} \ x : A . \ f | \langle X = A, v : A' \rangle | \langle v_0, \ldots, v_{n-1} \rangle \\
& \quad (\langle X = A, e : A' \rangle) \text{ open } e \text{ as } \langle X, x \rangle \text{ in } e' \\
& \quad \langle e_0, \ldots, e_{n-1} \rangle \text{ sel} [\langle e, e' \rangle] | e \ aop e' \\
& \quad e \ cop e' | \text{if} [A, A'] [e, X_1, e_1, X_2, e_2] \\
\end{align*}
\]

where \( n \in \mathbb{N} \)

\[
\begin{align*}
(fun) & \quad f ::= \lambda x : A . e | \lambda X : A . f \\
\text{(arith)} & \quad \text{aop} ::= + | \ldots \\
\text{(emp)} & \quad \text{cop} ::= < | \ldots
\end{align*}
\]

Figure 4: Syntax of the computation language \( \lambda_H \).

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

\[
E ::= \bullet | E \ e | v \ E | E [\ e | \langle X = A, E : A' \rangle | \text{open} \ E \text{ as } \langle X, x \rangle \text{ in } e | \text{open} \ v \text{ as } \langle X, x \rangle \text{ in } E | \langle v_0, \ldots, v_n, E, e_1, e_2, \ldots, e_{n-1} \rangle | \text{sel}[A][E, e] | v \text{ aop } e | v \text{ aop } E | E \text{ cop } e | v \text{ cop } E | \text{if}[A, A'] [E, X_1, e_1, X_2, e_2]
\]

The notation \( E[e] \) stands for the term obtained by replacing the hole \( \bullet \) in \( E \) by \( e \). The single step computation \( \rightsquigarrow \) relates \( E[e] \) to \( E[e'] \) when \( e \rightsquigarrow e' \) and \( \rightsquigarrow^* \) 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/IF. However an inspection of the rules shows that types are irrelevant for the evaluation, hence a type-erasure semantics, in which all type-related operations and parameters are erased, would be entirely standard.

**4.1 Static semantics**

The static semantics of \( \lambda_H \) shows the benefits of using a type language as expressive as TL. We can now define the type constructors of \( \lambda_H \) as constructors of an inductive kind \( \Omega \), instead of having them built into \( \lambda_H \). 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.

**Inductive \( \Omega : \text{Kind} \)**

\[
\begin{align*}
\text{snat} \quad & : \text{Nat} \to \Omega \\
\text{bool} \quad & : \text{Bool} \to \Omega \\
\to \quad & : \Omega \to \Omega \to \Omega \\
\text{tup} \quad & : \text{Nat} \to (\text{Nat} \to \Omega) \to \Omega \\
\forall \text{Kind} \quad & : \Pi \times \text{Kind}. (k \to \Omega) \to \Omega \\
\exists \text{Kind} \quad & : \Pi \times \text{Kind}. (k \to \Omega) \to \Omega \\
\forall \text{Kscm} \quad & : \Pi \times \text{Kscm}. (z \to \Omega) \to \Omega \\
\exists \text{Kscm} \quad & : \Pi \times \text{Kscm}. (z \to \Omega) \to \Omega
\end{align*}
\]

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

\[
\begin{align*}
A \to B \equiv & \ A \ B \\
\forall_{\mathcal{X}} X : A : B \equiv & \ A \ (\lambda X : A . \ B) \\
\exists_{\mathcal{X}} X : A := & \ A \ (\lambda X : A . \ B)
\end{align*}
\]

where \( s \in \{ \text{Kind}, \text{Kscm} \} \) and often drop the sort \( s \) when \( s = \text{Kind} \); e.g. the type void, containing no values, is defined as \( \forall_{\Omega} : \Omega \\equiv \forall_{\text{Kind}} \Omega \) (\( \Omega : \Omega \) t).

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

$$(\text{type env}) \quad \Gamma ::= \cdot | \Gamma, x : A$$

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 $T$ has type $\text{succ}$ zero in any valid environment. In rule E-NAT we use the metafunction $\hat{\cdot}$ to map natural numbers $n \in N$ to their representations as type terms. It is defined inductively by $\hat{0} = \text{zero}$ and $n + 1 = \text{succ} \hat{n}$, so $\Delta \vdash \hat{n} : \text{Nat}$ holds for all valid $\Delta$ and $n \in N$.

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

$$\begin{align*}
\text{nat} & : \Omega = \exists t : \text{Nat}. \text{snat} t \\
\text{bool} & : \Omega = \exists t : \text{Bool}. \text{bool} t
\end{align*}$$

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

$$\begin{align*}
\text{add} & = \lambda x_1 \cdot \text{nat}. \lambda x_2 \cdot \text{nat}. \\
& \quad \text{open} x_1 \cdot \langle t_1, x_1 \rangle \in \text{open} x_2 \cdot \langle t_2, x_2 \rangle \in \\
& \quad (t = t_1 + t_2, x_1 + x_2 : \text{snat} t) : \text{snat} t
\end{align*}$$

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:

Inductive List : Kind ::= nil : List  \\
| cons : $\Omega \rightarrow \text{List} \rightarrow \text{List}$

nth : List $\rightarrow \text{Nat} \rightarrow \Omega$

nth nil $= \lambda t : \text{Nat}. \text{void}$

nth (cons $t_1 \cdot t_2$) $= \lambda t : \text{Nat}. \text{if} \exists t \Omega t_1 \cdot (\text{nth} t_2)$

Thus nth $\pi$ 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 $A \cdot A' \equiv \text{cons} A \cdot A'$. The type of pairs is derived: $A \times A' \equiv \text{tup} 2 (\text{nth} (A \cdot A' \cdot \text{nil}))$. Thus for instance $\vdash \langle \overline{T2}, \overline{T} \rangle : \text{snat} \overline{T2} \times \text{snat} \overline{T}$ 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 Nat $\rightarrow$ Nat $\rightarrow$ Kind. Intuitively, LT represents a binary relation on kind Nat, so LT $\pi \pi$ 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 the axioms of the axioms $\forall v \in N. 0 < n + 1$ and $\forall m, n \in N. m < n \lor m + 1 < n + 1$.

Inductive LT : Nat $\rightarrow$ Nat $\rightarrow$ Kind

$\vdash \text{lts} : \Pi : \text{Nat}. \text{LT} \rightarrow \text{zero} (\text{succ} t)$

$\vdash \text{ltss} : \Pi : \text{Nat}. \Pi' : \text{Nat}. \text{LT} \rightarrow \text{LT} \rightarrow \text{LT} \rightarrow \text{LT} (\text{succ} t) (\text{succ} t')$

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 (defined later); this is sufficient since proof objects are never analyzed in $\lambda_H$, so the full power of elimination is not necessary for LT.

In the component selection construct sel$[A] \langle e, e' \rangle$ 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-erase semantics.

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

4.2 Example: bound check elimination

A simple example of the generation, propagation, and use of proofs in $\lambda_H$ 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 $\beta\eta$-convertible to the form vec $\pi A$ for

$$\begin{align*}
\text{vec} & : \Omega \rightarrow \Omega \rightarrow \Omega \\
\text{vec} & = \lambda t : \Omega. \lambda \pi : \Omega. \text{tup} t (\text{nth} (\text{repeat} t t'))
\end{align*}$$

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

$$\begin{align*}
\text{sumvec} & : \forall t : \text{Nat}. \text{snat} t \rightarrow \text{vec} t \text{nat} \rightarrow \text{nat} \\
& \equiv \lambda t : \text{Nat}. \lambda \pi : \text{snat} t. \lambda v : \text{vec} t \text{nat}. \\
& \quad (\text{fixloop} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}. \\
& \quad \lambda t : \text{Nat}. \lambda \sum : \text{nat}. \\
& \quad \text{open} i \cdot \langle t', i' \rangle \in \\
& \quad \text{if} \text{LTORtrue} t' t. \text{ltprf} t' t) \\
& \quad (i' < n, \\
& \quad t_1. \text{loop} (\text{add} i \langle T \rangle) \\
& \quad (\text{addsum} \langle \text{snat} t \rangle(v, i'))), \\
& \quad t_2. \text{sum})) \overline{\text{t0}}
\end{align*}$$

where

$$\begin{align*}
\text{LTORtrue} & : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \rightarrow \text{Kind} \\
\text{LTORtrue} & = \lambda t_1 : \text{Nat}. \lambda t_2 : \text{Nat}. \lambda lt : \text{Bool}. \text{Cond} t (\text{LT} t_1 t_2) \text{True} \\
\text{ltprf} & \text{of kind } \Pi' : \text{Nat}. \Pi : \text{Nat}. \text{LTORtrue} t' t (\text{lt} t' t) \text{ is a type term defined later.}
\end{align*}$$

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 $\Pi$ with respect to the less-than relation ensures that the type term $\text{ltprf} t' t$ represents a proof of the corresponding proposition at the type level, namely $\text{LT} t' t$. 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 $\lambda_H$ (Theorem 1) guaran-
Lemma 4 in Appendix A is the connection between the existence and subject reduction. A pivoting element in proving progress λ

### 4.3 Type safety

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

**Lemma 1 (Adequacy of the TL representation of arithmetic)**

1. For all \(m, n \in \mathbb{N}\), plus \(\overline{m}\overline{n}\equiv_{\beta \eta \iota} \overline{m+n}\).
2. For all \(m, n \in \mathbb{N}\), \(\text{lt}\overline{m}\overline{n}\equiv_{\beta \eta \iota} \text{true}\) if and only if \(m < n\).
3. For all \(m, n \in \mathbb{N}\), \(\text{lt}\overline{m}\overline{n}\equiv_{\beta \eta \iota} \text{true}\) if and only if there exists a type \(A\) such that \(\vdash A : \text{LT}\overline{m}\overline{n}\).

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

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

**Proof sketch** Follows from Lemmas 4 and 5 (Appendix A).
4.4 An example of proof generation

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

$\text{LT : Nat} \rightarrow \text{Nat} \rightarrow \text{Kind}$

$\text{LT} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\Pi R : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Kind}$. \\
$\Pi R = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. (\Pi : \text{Nat}. \text{R} \rightarrow (\text{succ} \ t) (\text{succ} \ t') \rightarrow R \ t')$

$\text{ltzs : Nat} \rightarrow \text{Nat} \rightarrow \text{Kind}$

$\text{ltzs} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{ltzs} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{ltss} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{ltss} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\

Next we define dependent conditionals on kinds $\text{Nat}$ and $\text{Bool}$.

$\text{dep_ifez : Nat} \rightarrow \text{Nat} \rightarrow \text{Kind}$

$\text{dep_ifez} \ k \ z t = \lambda k : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{dep_ifez} \ k \ z t = \lambda k : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{dep_ifez} \ (\text{succ} \ t) = \lambda k : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{dep_ifez} \ (\text{succ} \ t) = \lambda k : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\

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

$\text{LTcond : Nat} \rightarrow \text{Nat} \rightarrow \text{Kind}$

$\text{LTcond} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{LTcond} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\

$\text{LTimp : Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \rightarrow \text{Kind}$

$\text{LTimp} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{LTimp} = \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\

$\text{ltPrf : \Pi t' : \text{Nat}. \ \Pi t : \text{Nat}} \rightarrow \text{Nat} \rightarrow \text{Kind}$

$\text{ltPrf} = \lambda t' : \text{Nat}. \ \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\
$\text{ltPrf} = \lambda t' : \text{Nat}. \ \lambda t : \text{Nat}. \ \lambda \text{Lt} : \text{Nat}$. \\

5 CPS Conversion

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

We call the target calculus for this phase $\lambda K$, with syntax:

$\begin{align*}
\text{(val) } & \quad v ::= x \mid \pi t \mid \text{ff} | \langle X = A, v : A' \rangle | \langle v_0, \ldots, v_{n-1} \rangle \\
\text{(exp) } & \quad e ::= v[A_1, \ldots, A_n][v'] \mid \text{let } x = v \text{ in } e \\
& \quad \mid \text{let } (X, x) = \text{open } v \text{ in } e \\
& \quad \mid \text{let } x = \text{select}[A][v, v'] \text{ in } e \\
& \quad \mid \text{let } x = v \text{ ap } v' \text{ in } e \\
& \quad \mid \text{if } [A, A'][v, X_1, e_1, e_2] \\
\end{align*}$

Expressions in $\lambda K$ consist of a series of let bindings followed by a function application or a conditional branch. There is only one abstraction mechanism, fix, which combines type and value abstraction. Multiple arguments may be passed by packing them in a tuple. $\lambda K$ shares the TL type language with $\lambda M$. The types for $\lambda K$ all have kind $\Omega K$ which, as in $\lambda M$, is an inductive kind defined in TL. The $\Omega K$ kind has all the constructors of $\Omega$ plus one more (func). Since functions in CPS do not return values, the function type constructor of $\Omega K$ has a different kind:

$\rightarrow : \Omega K \rightarrow \Omega K$

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

$\text{func} : \Omega K \rightarrow \Omega K$

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

Typed CPS conversion involves translating both types and computation terms. Existing algorithms [21, 28] 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 $\lambda M$ expression $(X = A, e : B)$ to CPS, assuming that it has type $\exists X'. A'. B'$. We use $\text{Ktyp}$ to denote the meta-level translation function for the type language and $\text{Kexp}$ for the computation language. Under existing systems, the translation also transforms the witness $A$:

$\text{Kexp} [\langle X = A, e : B \rangle] = \\
\lambda k : \text{Ktyp} \exists X'. A'. B'. \text{Kexp} e (\lambda x : \text{Ktyp} [A/X] B). \text{Kexp} e (\lambda x : \text{Ktyp} [A/X] B).

k \langle X = \text{Ktyp} [A], x : \text{Ktyp} [B] \rangle$

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

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

$\text{Kexp} [\langle X = A, e : B \rangle] = \\
\lambda k : \text{Ktyp} \exists X'. A'. B'. \text{Kexp} e (\lambda x : \text{Ktyp} [A/X] B).

k \langle X = A, x : \text{Ktyp} [B] \rangle$

To preserve typing, we have to make sure that the two types $\text{Ktyp} [A/X] B'$ and $\text{Kexp} [A/X] (\text{Ktyp} [B])$ are equivalent. This seems impossible to achieve if $\text{Ktyp}$ is defined at the meta level.
Our solution is to internalize the definition of $K_{\Omega p}$ in our type language. We replace $K_{\Omega p}$ 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 $\text{Elim}$ construct.

\[
\begin{align*}
K (\text{snat } t) &= \text{snat } t \\
K (\text{bool } t) &= \text{bool } t \\
K (t_1 \rightarrow t_2) &= \mu \text{func } (K(t_1) \times K(t_2)) \\
K (\text{tup } t_1 t_2) &= \mu \text{tup } t_1 (\lambda t : \text{Nat}. K(t_2) t) \\
K (\text{Psi}_{\text{kind}} k t) &= \mu \text{func } (\lambda x : t. K(x k \rightarrow \bot)) \\
K (\text{Psi}_{\text{kasm}} z t) &= \mu \text{func } (\lambda x : z. K(z k \rightarrow \bot)) \\
K (\text{Ext}_{\text{kind}} k t) &= \mu \text{func } (\lambda x : k. K(t k)) \\
K (\text{Ext}_{\text{kasm}} z t) &= \mu \text{func } (\lambda x : z. K(t k)) \\
K \equiv & \lambda t : \Omega. \text{func } (K(t) \rightarrow \bot)
\end{align*}
\]

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

6 Closure Conversion

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

Our approach to closure conversion is based on Morrisett et al. [28], who adopt a type-erasure interpretation of polymorphism. We use the same idea for existential types. The language that we use for this phase is called $\lambda C$ with syntax:

\[
\begin{align*}
(\text{val}) \quad v &::= x | \pi \times tt | ff \mid \text{fix } x'. \Pi \times \ldots \Pi \times x_n. A_n(x : A) \mid e \mid v[A] \mid (v_0, \ldots, v_{n-1}) \mid (X = A, v : A') \\
(\text{exp}) \quad e &::= v v' \mid let x = v in e \mid let x = selt[A](v, v') in e \mid let (X, x) = open e in e \mid let x = v aop v' in e \mid let x = v cop v' in e \mid if [B, A](v, X_1, c_1, X_2, c_2)
\end{align*}
\]

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

7 Related Work

Our type language is a variant of the calculus of constructions [10] extended with inductive definitions (with both small and large elimination) [35, 41]. We omitted parameterized inductive kinds and dependent large elimination to simplify our presentation, however, all our meta-theoretic proofs carry over to a language that includes them. We support $\eta$-reduction in our language while the official Coq system does not. The proofs for the properties of TL are adapted from Werner [41] and Geuvers [16]; the main difference is that our language has kind-schema variables and a new product formation rule (Ext, Kind) which are not in Werner’s system.

The Coq proof assistant provides support for extracting programs from proofs [35]. 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 compilation terms. In fact, the logic in our type language does not have to be constructive; there is no problem with adding classical reasoning to our proof system.

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

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

We have discussed the relationship between our work and those on PCC, typed assembly languages, and intensional type analysis in Section 1. Inductive definitions subsume and generalize earlier systems on intensional type analysis [22, 13, 39]; the type-analysis construct in the computation language can be eliminated using the technique proposed by Crary et al. [15].

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

8 Conclusions

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

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

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

Acknowledgment

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

References

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

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

We also need a proposition guaranteeing that equivalence of constructor applications implies equivalence of their arguments; it is a corollary of the confluence of TL (Theorem 76).

**Lemma 3** If $\text{Ctor}(i, I) \equiv A_{\beta\eta\iota}$ $\text{Ctor}(i', I') \equiv A'$, then $i = i'$ and $I \equiv A_{\beta\eta\iota}$ $I' \equiv A'$.

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

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

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

If $e = \text{sel}[A'[[v, v']$, by inspection of the typing rules the derivation of $\vdash \triangledown : A$ 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

$$\vdash \triangledown : \text{tup} A_2 A''$$

$$\vdash \triangledown : \text{snat} \hat{m}$$

$$\vdash A' : \text{LT} A_1 A_2$$

$$\vdash \text{sel}[A'[[v, v']] : A''A_1$$

$$\vdash \text{sel}[A'[v, v'] : A$$

where $A \equiv_{\beta\eta\iota} A''A_1$. By inspection of the typing rules, rules other than E-CONV assign to all values types which are applications of constructors of $\Omega$. Since the derivation $D$ 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 $\text{tup} A_2 A''$. Then by Lemma 3 this type must be an application of $\text{tup}$, and again by inspection the only rule which applies is E-TUP, which implies $v = (v_0, \ldots, v_{n-1})$, and the derivation $D$ must have the form

$$\forall i < n \quad D_i \vdash v_i : A''_i \quad \vdash (v_0, \ldots, v_{n-1}) : \text{tup} \hat{m} A''_1$$

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

**Lemma 5 (Subject Reduction)** If $\vdash \triangledown : e : A$ and $e \triangledown e'$, then $\vdash \triangledown : e' : A$.

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

- The derivation for $e = \text{sel}[A'[v, v']]$ in standard form has the shape

$$\forall i < n \quad D_i \vdash v_i : A''_i \quad \vdash A' : \text{LT} A_1 A_2$$

$$\vdash \text{sel}[A'[[v, v']] : A''_1 \quad \vdash \text{sel}[A'[v, v'] : A$$

where $A \equiv_{\beta\eta\iota} A''_1$, $A_1 \equiv_{\beta\eta\iota} A''_1$, and $A_2 \equiv_{\beta\eta\iota} \hat{m}$. Since $e \triangledown e'$ only by rule R-SEL, we have $m < n$ and $e' = v_m$, so from $D_m$ and $A_1 \equiv_{\beta\eta\iota} A''_1 = \beta\eta\iota$ we obtain a derivation of $\vdash \triangledown : e' : A$.

- In the case of if the standard derivation $D$ of

$$\vdash \triangledown : \text{if}[B, A']([[tt, X_1, e_1, X_2, e_2] : A$$

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 $\hat{E}$ of $\vdash A' : B A''$, and $A'' \equiv_{\beta\eta\iota} \text{true}$ (since rule E-TRUE assigns bool true to tt), hence we have $\vdash A' : B$ true by CONV. By Lemma 2 from $E$ and the derivation of $X_1 : B$ true; $\vdash e_1 : A$ (provided as another premise), since $X_1$ is not free in $A$ (ensured by the premise $\vdash A : \Omega$) we obtain a derivation of $\vdash \triangledown : [A'/X_1]e_1 : A$.

**B CPS Conversion (Details)**

We start by defining a version of $\lambda_H$ 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.

$$\text{(exp)} \quad e ::= e^A$$

$$\bar{e} ::= x | \bar{p} | \bar{tt} | \bar{ff} | f | \text{fix } x : A. f | e e' | e[A]$$

$$| (X = A, e: A') : \text{open c as } (X, x') \text{ in } e'$$

$$| (e_0, \ldots, e_{n-1}) : \text{sel}[A'](e, e') \quad e aop e'$$

$$| e \cop e' | \text{if}[A', A']'(e, X_1, e_1, X_2, e_2)$$

$$\text{(fun)} \quad f ::= \bar{f}^A$$

$$\bar{f} ::= \lambda x : A. e | AX : A. f$$
The target language $\lambda_K$ of the CPS conversion stage has been defined in Section 5. We use the following syntactic sugar to denote non-recursive function definitions and value applications in $\lambda_K$ (here $x'$ is a fresh variable):

$$
\lambda x : A. e \equiv \text{fix } x'[[x : A). e]
$$

$$
v v' \equiv \text{fix } x'[x_1 : A_1, \ldots, x_n : A_n](x : A). e
$$

$$
\Delta X_1 : A_1, \ldots, \Delta X_n : A_n, \lambda x : A. e \equiv \text{fix } x'[X_1 : A_1, \ldots, X_n : A_n](x : A). e
$$

In the static semantics of $\lambda_K$, we use two forms of judgments. As in $\lambda_H$, the judgment $\Delta ; \Gamma \vdash_k v : A$ indicates that the value $v$ is well formed and of type $A$ in the type and value contexts $\Delta$ and $\Gamma$ respectively. Moreover, $\Delta ; \Gamma \vdash_k e$ indicates that the expression $e$ is well formed in $\Delta$ and $\Gamma$. In both forms of judgments, we omit the subscript from $\vdash_k$ it when it can be deduced from the context.

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

(K-FIX) for all $i \in \{1, \ldots, n\}$,

$$
\Delta ; \Gamma \vdash_k A_i : s_i
\Delta, X_1 : A_1, \ldots, X_n : A_n \vdash : \Omega
\Delta, X_1 : A_1, \ldots, X_n : A_n, \Gamma, x : A' \vdash : e
$$

$$
\Delta; \Gamma \vdash_k \text{fix } x'[X_1 : A_1, \ldots, X_n : A_n](x : A). e : A'
$$

(K-APP)

$$
\Delta; \Gamma \vdash_k v : [A_1/X_1] \ldots [A_n/X_n] A
\Delta; \Gamma \vdash_k v : A
\Delta; \Gamma, x : A \vdash : e
$$

(K-VAL)

$$
\Delta; \Gamma \vdash_k v : \text{tup } A'' \ B
\Delta; \Gamma \vdash_k v' : \text{snat } A'
\Delta; \Gamma \vdash_k \text{let } x = v \text{ in } e
$$

(K-SEL)

$$
\Delta; \Gamma \vdash_k v : \exists y : [X/Y] A \vdash : e
\Delta; \Gamma \vdash_k \text{let } (x, y) = \text{open } v \text{ in } e \quad (X \notin \Delta)
$$

(K-OPEN)

$$
\Delta; \Gamma \vdash_k v : \text{snat } A
\Delta; \Gamma \vdash_k v' : \text{snat } A'
\Delta; \Gamma, x : \text{snat } (\text{plus } A \ A') \vdash : e
$$

(K-ADD)

$$
\Delta; \Gamma \vdash_k x = v + v' \text{ in } e
$$

(K-LT)

$$
\Delta; \Gamma \vdash_k x = v \ L T \ A' \ A'
\Delta; \Gamma, x : \text{bbool } (\text{lt } A' \ A') \vdash : e
$$

(K-IF)

$$
\Delta; \Gamma \vdash_k x = v \ L T \ A' \ A'
\Delta; \Gamma \vdash_k x = v \ L T \ A' \ A'
$$

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

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

\[ \mathcal{K}_{\text{fix}}[(\lambda x : A. e).B] = \lambda x_{\text{arg}} : K(A) \times K_{\text{fix}}(B). \]

\[ \text{let } x = \text{sel}[\text{tPrf } \delta](x_{\text{arg}}, 0) \text{ in } \]

\[ \text{let } k = \text{sel}[\text{tPrf } \bar{\delta}](x_{\text{arg}}, \Gamma) \text{ in } \]

\[ \mathcal{K}_{\text{exp}}[e^A] k \]

Figure 6: CPS conversion: from $\lambda_H$ to $\lambda_K$.

**Proposition 2 (Type Correctness of CPS Conversion)**

If $\vdash_k e : A$, then $\vdash_k \mathcal{K}_{exp}[e^A] : \text{func } (K(A) \rightarrow \bot)$. 

C Closure Conversion (Details)

The main difference in the static semantics between \(\lambda_K\) and \(\lambda_C\) 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 \(\lambda_C\) are standard.

\[
\begin{align*}
\Delta; \Gamma \vdash v_1 : A \rightarrow \bot & \quad \Delta; \Gamma \vdash v_2 : A & \quad \Delta; \Gamma \vdash v_1 \circ v_2 : A & \\
\end{align*}
\]

(C-tapp)

\[
\begin{align*}
\Delta; \Gamma \vdash x' : \forall \alpha . X : A & \quad \Delta; \Gamma \vdash e : B & \quad \Delta; \Gamma \vdash \text{Ctor}(x', e) : A & \\
\end{align*}
\]

(C-app)

The definition of the closure transformation for the computation terms of \(\lambda_K\) is given in Figure 7.

Proposition 3 (Type Correctness of Closure Conversion)

If \(\Delta; \Gamma \vdash v : A\), then \(\Delta; \Gamma \vdash C_{\text{val}}[v] : \text{Cl}(A)\)."
Definition 6 Let C be a well-formed constructor kind for X. Then C is of the form \( \Pi Y: \hat{A}: X \). If all the \( Y \)'s are \( t \)'s, that is, \( C \) is of the form \( \Pi Y: \hat{A}: X \), then we say that \( C \) is a small constructor kind (or just small constructor when there is no ambiguity) and denote it as \textit{small}(C).

Our inductive definitions reside in \textit{Kind}, whereas a small constructor does not make universal quantification over universes of type \textit{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 \cite{17}. In particular, we restrict large elimination to inductive definitions with only small constructors.

Next, we define the set of reductions on our terms. The definition of \( \beta \) and \( \eta \)-reduction is standard. The \( \iota \)-reduction defines primitive recursion over inductive objects.

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

\[
\begin{align*}
\Phi_{X,I,B'}(X, A') & \overset{\text{def}}{=} A' \\
\Phi_{X,I,B'}(\Pi Y: B, C', A') & \overset{\text{def}}{=} \lambda Y: B. \Phi_{X,I,B'}(C', A', Y) \\
\Phi_{X,I,B'}(\Pi Y: B, I) & \overset{\text{def}}{=} \lambda Z: (\Pi Y: B, I). \Phi_{X,I,B'}(C', A', Z) (\lambda Y: B. B'(Z Y))
\end{align*}
\]

Definition 8 The reduction relations on our terms are defined as:

\[
(\lambda X: A. B) A' \leadsto_{\beta} [A'/X]B \\
\lambda X: A. (B X) \leadsto_{\eta} B, \text{ if } X \notin \mathbb{FV}(B) \\
\text{Elim}[I, A''](\text{Ctor}(i, I) \hat{A})(\bar{B}) \leadsto_{\iota} (\Phi_{X,I,B'}(C_i, B_i)) \hat{A}
\]

where \( I = \text{Ind}(X: \text{Kind}(\hat{C})) \)

By \( \triangleright_{\beta} \), \( \triangleright_{\eta} \), and \( \triangleright \), we denote the relations that correspond to the rewriting of subterms using the relations \( \sim_{\beta} \), \( \sim_{\eta} \), and \( \sim_{\iota} \) respectively. We use \( \sim \) and \( \triangleright \) for the unions of the above relations. We also write \( \triangleright^* \) and \( \triangleright^+ \) (respectively \( \triangleright^*_\iota \) etc.) for the reflexive-transitive and transitive closures of \( \triangleright \) (respectively \( \triangleright_{\iota} \) etc.) and \( \sim_{\beta|\eta} \) for the reflexive-symmetric-transitive-closure of \( \triangleright \).

We say that a sequence of terms \( A_1, \ldots, A_n \), such that \( A \triangleright A_1 \triangleright A_2 \ldots \triangleright A_n \), is a chain of reductions starting from \( A \).

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

\[
\text{Elim}[I, A''(\hat{A})/B] \leadsto_{\iota} B_i
\]

In the case when \( C_i = \Pi Y: B, X \) where \( X \) does not occur free in \( B \), then \( A \) must be in the form Ctor(\( i, I \), \( \hat{A} \) with \( A_i \) of type \( B_i \).

None of the arguments are recursive. Therefore, the Elim should just select the \( B_i \) branch and pass the constructor arguments to it. Accordingly, the reduction yields (by expanding the \( \Phi \) macro):

\[
\text{Elim}[I, A''(\hat{A})/B] \leadsto_{\iota} B_i \hat{A}
\]

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

\[
\text{Elim}[I, A''(\hat{A})/B] \leadsto_{\iota} B_i A_1 (\lambda Y: B'. \text{Elim}[I, A''(\hat{A})/B]) A_2 \ldots A_n
\]

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.

Definition 9 defines the \( \Psi \) macro which represents the type of the large Elim branches. Definition 10 defines the \( \zeta \) macro which represents the type of the small elimination branches. The different cases follow from the \( \iota \)-reduction rule in Definition 8.

Definition 9 Let \( C \) be a well-formed constructor kind for \( X \) and let \( A', I \) and \( B' \) be terms. We define \( \zeta_{X,I}(C, A') \) recursively based on the structure of \( C \):

\[
\begin{align*}
\zeta_{X,I}(X, A') & \overset{\text{def}}{=} A' \\
\zeta_{X,I}(\Pi Y: B, C', A') & \overset{\text{def}}{=} \Pi Y: B. \zeta_{X,I}(C', A') \\
\zeta_{X,I}(A \rightarrow C', A') & \overset{\text{def}}{=} [I/X]A \rightarrow [A'/X]A \rightarrow \Psi_{X,I}(C', A')
\end{align*}
\]

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

Definition 10 Let \( C \) be a well-formed constructor kind for \( X \) and let \( A', I \), and \( B' \) be terms. We define \( \zeta_{X,I}(C, A', B') \) recursively based on the structure of \( C \):

\[
\begin{align*}
\zeta_{X,I}(X, A', B') & \overset{\text{def}}{=} A' B' \\
\zeta_{X,I}(\Pi Y: B, C', A', B') & \overset{\text{def}}{=} \Pi Y: B. \zeta_{X,I}(C', A', B') \\
\zeta_{X,I}(\Pi Y: B, X \rightarrow C', A', B') & \overset{\text{def}}{=} \Pi Y: B, \hat{B}. \hat{B}.'(A'(Z Y)) \rightarrow \zeta_{X,I}(C', A', B' Z)
\end{align*}
\]

where \( X \) is not free in \( B \) and \( B' \).

Definition 11 We use \( \Delta_{i,k} \) to denote that the environment does not contain any \( z \) variables.

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

\[
\begin{align*}
\vdash \text{Kind} : \text{Kscm} & \quad (AX1) \\
\vdash \text{Kscm} : \text{Ext} & \quad (AX2) \\
\Delta \vdash C : \text{Kind} & \quad \Delta, t : C \vdash A : B & \quad (\text{WEAK1}) \\
\Delta \vdash C : \text{Kscm} & \quad \Delta, k : C \vdash A : B & \quad (\text{WEAK2}) \\
\Delta \vdash C : \text{Ext} & \quad \Delta, z : C \vdash A : B & \quad (\text{WEAK3}) \\
\Delta, X : A \vdash B : B' & \quad \Delta \vdash \Xi : A, B : s & \quad (\text{FUN}) \\
\Delta \vdash A : B \vdash B : B' & \quad \Delta \vdash A : B \vdash B : B' & \quad (\text{APP}) \\
\Delta \vdash A : s_1 & \quad \Delta, X, A \vdash B : s_2 & \quad (s_1, s_2) \in R & \quad (\text{PROD})
\end{align*}
\]
for all $i$ \[ \Delta, X : \text{Kind} \vdash C_i : \text{Kind} \quad \text{ufc}_X(C_i) \] (IND)

\[ \Delta \vdash \text{Ind}(X : \text{Kind}) \{ \vec{C} \} : \text{Kind} \] (CON)

\[ \Delta \vdash I : \text{Kind} \quad \text{where} \quad I = \text{Ind}(X : \text{Kind}) \{ \vec{C} \} \]

\[ \Delta \vdash \text{Ctor}(i, I) : [I/X]C_i \] (ELIM)

\[ \Delta \vdash \text{Elim}[I, A'] \{ \vec{B} \} : A' \quad A \]

where \[ I = \text{Ind}(X : \text{Kind}) \{ \vec{C} \} \]

\[ \Delta \vdash A : I \quad \Delta \vdash A' : I \to \text{Kind} \]

for all $i$ \[ \Delta \vdash B_i : \xi_{X,i}(C_i, A', \text{Ctor}(i, I)) \] (ELIM)

\[ \Delta \vdash \text{Elim}[I, A'] \{ \vec{B} \} : A' \quad A \]

where \[ I = \text{Ind}(X : \text{Kind}) \{ \vec{C} \} \]

\[ \Delta \vdash A : B \]

\[ \Delta \vdash B : s \quad \Delta \vdash B : s \quad B =_{\beta\eta} B' \] (CONV)

\[ \Delta \vdash A : B' \]

\[ \| s \| = s \]

\[ \| X \| = X \]

\[ \| A_1 \cup A_2 \| = \| A_1 \| \cup \| A_2 \| \]

\[ \| \lambda X : A_1 \cup A_2 \| = \lambda X : \| A_1 \| \cup \| A_2 \| \]

\[ \| \Pi X : A_1 \cup A_2 \| = \Pi X : \| A_1 \| \cup \| A_2 \| \]

\[ \| \text{Ind}(X : \text{Kind}) \{ A \} \| = \| \text{Ind}(X : \text{Kind}) \{ \| A \| \} \| \]

\[ \| \text{Ctor}(i, A_1) \| = \| \text{Ctor}(i, A_1) \| \]

\[ \| \text{Elim}[I, A_2](A_1) \| = \| \text{Elim}[I, A_2](\| A_1 \|) \| \]

\[ \| \text{Ind}(X : \text{Kind}) \{ A \} \| = \| \text{Ind}(X : \text{Kind}) \{ \| A \| \} \| \]

Lemma 6 For all terms $A, B, A', B'$, and for all variables $X$ and $Y$, we have that $[\lambda Y : A', B/X]A =_{\beta_\eta} [\lambda Y : B', B/X]A$.


\[ \Box \]

Lemma 7 For all terms $A$, we have $A =_{\beta_\eta} A$.

Proof Follows from lemma 6.

\[ \Box \]

Definition 12 ($\iota_0$ reduction) We say that $A \triangleright_\iota_0 A'$ iff $A \triangleright_\iota A'$ and $A \neq A'$.

Proposition 13 For all terms $A$ and $A'$, if $A \triangleright_\beta A'$, then $A \triangleright_\iota_0 A'$ or $\| A \| = \| A' \|$. Similarly, if we have that $A \triangleright_\beta A'$, then $A \triangleright_\iota_0 A'$ or $\| A \| = \| A' \|$. Moreover, if $\| A \| \triangleright_\beta \eta A'$ and $\| A' \| = \| A' \|$, then there exists a $A''$ such that $A \triangleright_\eta A''$ and $\| A'' \| = \| A' \|$.

Lemma 8 (Confluence for unmarked terms) For all unmarked terms $\| A \|$, the $\beta\eta\iota_0$ reduction is confluent.

The proof is based on the method of parallel reductions due to Tait and Martin-Löf.

Definition 14 (Parallel reduction) Define $\leftrightarrow$ on unmarked terms as below, in which we assume that $A \rightarrow A'$, $B \rightarrow B'$, etc.

\[ A \leftrightarrow A \]

\[ A \rightarrow A' \quad B \rightarrow B' \]

\[ \lambda X : A \rightarrow \lambda X : A' \]

\[ \Pi X : A \rightarrow \Pi X : A' \]

\[ \text{Ind}(X : \text{Kind}) \{ A \} \rightarrow \text{Ind}(X : \text{Kind}) \{ A' \} \]

\[ \text{Ctor}(i, I) \rightarrow \text{Ctor}(i, I') \]

\[ \text{Elim}[A, C] \{ I \} \{ A \} \rightarrow \text{Elim}[A, C] \{ I' \} \{ A' \} \]

\[ \lambda X : \ldots A B \rightarrow [B' / X]A' \lambda X : \ldots A \rightarrow A' \quad \text{if} \quad X \notin \text{FV}(A) \]

\[ \text{Elim}[I, C] \{ (\text{Ctor}(i, I') B) \} \rightarrow \Phi_{X,i', B'} \{ C_i', A_i' \} \quad B' = \lambda Y : \ldots (\text{Elim}[I', C'](Y) \{ A' \}) \]

where $I = \text{Ind}(X : \text{Kind}) \{ A' \}$

The parallel reduction commutes with respect to substitution.

Lemma 9 If $A \rightarrow A'$ and $B \rightarrow B'$, then $[B / X]A \rightarrow [B' / X]A'$.

Proof By induction over the fact that $A \rightarrow A'$.

The parallel reduction also has the following properties with respect to terms such as products and inductive definitions. The proof in each case is immediate and follows by induction over the structure of the term.
**Proposition 15** Suppose $A = \Pi X : \vec{B} . Y \vec{C}$. If $A$ can be reduced to $A'$ through a reduction relation ($\rightarrow_{\cdot}$, etc.), then $A'' = \Pi X : \vec{B}' . Y \vec{C}'$ where all the $\vec{B}$ and $\vec{C}$ can be reduced to $\vec{B}'$ and $\vec{C}'$ by the same reduction relation.

**Proposition 16** Suppose $A = \Pi X : \vec{B} . Y \vec{C}$ and $A' = \Pi X : \vec{B}' . Y \vec{C}'$ be two terms such that both can be reduced to $A''$ through a reduction relation ($\rightarrow_{\cdot}$, etc.). Then $A'' = \Pi X : \vec{B}'' . Y \vec{C}''$ where $\vec{B}$ and $\vec{C}$ can be reduced to $\vec{B}''$ by the same relation and $\vec{C}$ and $\vec{C}'$ can be reduced to $\vec{C}''$ by the same relation.

The parallel reduction is important because it subsumes the single step reduction: that is, if $A \rightarrow A'$, then we have that $A \rightarrow A''$ which also implies that $A \rightarrow A'$. From here, to show the confluence of $\rightarrow$, it suffices to show the confluence of parallel reduction.

**Lemma 10** For all unmarked terms $D$, $D'$, $D''$, we have that if $D \rightarrow D'$ and $D \rightarrow D''$, then there exists a $D'''$ such that $D' \rightarrow D'''$ and $D'' \rightarrow D'''$.

**Proof** The proof is by induction over the structure of $D$. We will only show one case here.

- Suppose $D = \text{Elim}[I, C]\{(\text{Ctor} (i, I) \vec{B})\}\{\vec{A}\}$.

  - We can then have $D' = (\Phi_{X, I', B'}(C_i', A_i')) \vec{B}'$ and $D'' = (\Phi_{X, I'', B'}(C_i'', A_i'')) \vec{B}''$. We have that $I' = \text{Ind}(X : \text{Kind})\{\vec{C}_i\}$ and $I'' = \text{Ind}(X : \text{Kind})\{\vec{C}_i''\}$. This implies that $C_i \rightarrow C_i'$ and $C_i \rightarrow C_i''$. By applying the induction hypothesis to the subterms, we get that $I' \rightarrow I'''$ and $I'' \rightarrow I'''$ and so on for the other subterms. From here and proposition 16, it follows that we can take $D''' = (\Phi_{X, I''' , B'}(C_i', A_i')) \vec{B}'''$.

- Suppose $D' = \text{Elim}[I', C']\{(\text{Ctor} (i, I') \vec{B}')\}\{\vec{A}'\}$ and $D'' = (\Phi_{X, I'', B'}(C_i'', A_i'')) \vec{B}''$. As above we can again define $I'''$, $C_i'''$, etc. and take $D''' = (\Phi_{X, I''' , B'}(C_i', A_i')) \vec{B}'''$.

- Also $D' = \text{Elim}[I', C']\{(\text{Ctor} (i, I') \vec{B}')\}\{\vec{A}'\}$ and $D'' = \text{Elim}[I'', C''\{(\text{Ctor} (i, I'') \vec{B}'')\}\{\vec{A}''\}$. In this case, we can again take that $D''' = \text{Elim}[I''' , C''\{(\text{Ctor} (i, I'')) \vec{B}''\}\{\vec{A}''\}$.

As a corollary of the confluence of unmarked terms we get the following:

**Corollary 17** If $A$ and $B$ are two distinct sorts or two distinct variables or a variable and a sort, then we have that $A \neq B$.

We will need another lemma – that of the delay of $\eta$ reduction. But before that, we have to define another variant of the $\iota$ reduction. This essentially says that a $\iota$ reduction that would appear only after a series of eta reductions can be reduced straightforwardly without going through the eta reductions. For well typed terms, this is equivalent to $\iota$ reduction, but it also allows us to retain the property of delay of $\eta$ reduction for ill-typed terms.

**Proposition 18** For all terms $A_1$ and $A_2$, we have that $A_1 \equiv_{\beta \eta} A_2$ if and only if $A_1 \equiv_{\beta \eta} A_2$.

**Lemma 11** If $A \equiv_{\eta} A' \equiv_{\beta \iota} A''$, then either $A \equiv_{\beta \eta} A'$, or there exists a $A'''$ such that $A \equiv_{\beta \eta} A''' \equiv_{\eta} A''$.

**Proof** The proof is by induction over the structure of $A$. We will consider only the cases that do not follow directly from the induction hypothesis.

- If $C \equiv_{\eta} C'$, then it follows immediately from the induction hypothesis.
- If $D \equiv_{\eta} D'$ and $C = \lambda X : B : B'$ and $A'' = [D' / X] B'$, then take $A''' = [D / X] B'$. The other cases follow from the induction hypothesis.

- $A = \lambda X : C . B X$. Suppose $A'' = B'$ where $B \equiv_{\beta \iota} B'$. But then we also have that $A \equiv_{\beta \eta} \lambda X : C . B X$. Since the reduction does not introduce new free variables, this term can now $\eta$-reduce to $B'$.

**Lemma 12 (Delay of $\eta$ reduction)** For all terms $A$ and $A'$, if $A \equiv_{\iota} A'$, then there exists a term $A''$ such that $A \equiv_{\beta \eta} A'' \equiv_{\eta} A'$.

**Proof** Follows from lemma 11.

We will next prove Geuvers’ lemma which is essentially a weak form of confluence. This is enough to prove the uniqueness of types and subject reduction. But before that we need to define the counterpart of the $\iota$ reduction for unmarked terms. We define it in the obvious way.

**Definition 19 ($\iota_0$ reduction)** We say that $A \equiv_{\iota_0} A'$ iff $A \equiv_{\iota} A'$ and $\|A\| \equiv_{\|A'\|}$.

As before it has the following property:

**Proposition 20** Suppose $A \equiv_{\iota} A'$. Then either $\|A\| \equiv_{\|A'\|}$, or $\|A\| \equiv_{\iota_0} \|A'\|$. Moreover, if $\|A\| \equiv_{\iota_0} \|A'\|$, then $A \equiv_{\iota} A'$.

**Lemma 13 (Geuvers lemma)**

- If $A = \beta \eta X \vec{A}$, then $A \equiv_{\beta \eta} \lambda \vec{X} \vec{A}. (X \vec{B} \vec{C})$ where for all $i$, $A_i = \beta \eta B_i$ and for all $j$, $C_j \equiv_{\eta} Y_j$.
- If $A = \beta \eta \Pi X : A_1 . A_2$, then $A \equiv_{\beta \eta} \lambda \vec{X} . \vec{A}. (\Pi X : A_3 . A_4) \vec{B}$ where $A_1 = \beta \eta A_1$ and $A_2 = \beta \eta A_2$ and for all $i$, $B_i \equiv_{\eta} Y_i$.
- If $A = \beta \eta \text{Ctor} (i, I) \vec{C}$, then $A \equiv_{\beta \eta} \lambda \vec{Y} . \vec{A}. (\text{Ctor} (i, I') \vec{C}') \vec{B}$ where for all $i$, $C_i = \beta \eta C_i'$ and for all $j$, $B_j \equiv_{\eta} Y_j$, and $I = \beta \eta I'$.
We partition the set of terms into four classes: the

Definition 21

D.2.2 Classification of terms

Proof

The proof for each of the cases is similar and is by induc-

tion over the length of the equivalence relation. We will show only

one case here.

• Suppose \( A =_{\beta_{\eta}} X \ A. \) By the induction hypothesis, there

exists an \( A'' \) such that

\[
A'' \triangleright_{\beta_{\eta}} \lambda Y : A'. (X \ B \ C)
\]

and \( A \triangleright_{\beta_{\eta}} A'' \) or \( A'' \triangleright_{\beta_{\eta}} A. \)

– The case where \( A \triangleright_{\beta_{\eta}} A'' \) is immediate.

– The case where \( A \triangleright_{\eta} A'' \) follows from the lemma of
delay of \( \eta \)-reduction.

– If \( A'' \triangleright_{\beta_{\eta}} A, \) then the required result follows from the
confluence of \( \beta_{\eta} \)-reduction.

– Suppose \( A'' \triangleright_{\eta} A. \) Then from the confluence of
\( \beta_{\eta} \)-reduction on unmarked terms, we get that \( A \triangleright_{\eta} X \ B \ D \)
where \( B \triangleright_{\beta_{\eta}} D, \) from the lemma of
delay of \( \eta \)-reduction, we get that

\[
\| A \| \triangleright_{\beta_{\eta}} \lambda Y : \_ X \ D \widetilde{F} \triangleright_{\eta} \_ X \ D
\]

From proposition 20 we can deduce the existence of

a term \( A_1 \) such that \( A \triangleright_{\beta_{\eta}} A_1 \) and \( \| A_1 \| = \lambda Y : \_ X \ D \widetilde{F}. \) The required result follows from here.

\[ \square \]

D.2.3 Well typed terms

We now consider the well typed terms. The following lemmas are
proved easily by induction over the typing derivations.

Lemma 15 (Substitution) If we can derive
\( \Delta_1 , (X, A) , \Delta_2 \vdash B : C \) and \( \Delta_1 , \vdash A : A \),
then we can derive \( \Delta_1 , ([A_2 / X] \Delta_2) \vdash [A_2 / X] B : [A_2 / X] C. \)

Proof

Straightforward induction over the structure of the derivation.

Lemma 16 If we can derive \( \Delta_1 , (X, A) , \Delta_2 \vdash B : C \),
then we also have that \( \Delta_1 \vdash A : s \) for some sort \( s \).
Moreover, we also have that \( \Delta_1 , (X, A) , \Delta_2 \vdash A : s. \)

Proof

The proof is by induction over the structure of the derivation.

Lemma 17 If we have that \( \Delta \vdash I L X : A : s \),
then we have that \( \Delta, X : A \vdash B : s. \)

Proof

The only interesting case is for the \( \text{CONV} \) case which
follows from Corollary 17.

Lemma 18 If the judgment \( \Delta \vdash A : B \) is derivable,
then either \( B = \text{Ext}, \) or \( \Delta \vdash B : s \) for some sort \( s. \)

Proof

The proof is a straightforward induction over the structure of the derivation.

Lemma 19 (Inversion) If the judgment \( \Delta \vdash A : B \) is derivable,
then

\[
A = t \quad \Rightarrow \quad t \in \Delta
\]
\[
A = k \quad \Rightarrow \quad k \in \Delta
\]
\[
A = z \quad \Rightarrow \quad z \in \Delta
\]
\[
A = \text{Kind} \quad \Rightarrow \quad B = \beta \eta i \text{Kscm}
\]
\[
A = \text{Kscm} \quad \Rightarrow \quad B = \text{Ext}
\]
\[
A = \Pi X:A_1. A_2 \quad \Rightarrow \quad \Delta \vdash A_1 : s_1
\]
\[
A = \lambda X:A_1. A_2 \quad \Rightarrow \quad \Delta \vdash \lambda X : A_1 \vdash A_2 : A_3
\]
\[
A = A_1 A_2 \quad \Rightarrow \quad \Delta \vdash A_1 : \Pi X : A_3
\]
\[
A = \text{Ind}(X: \text{Kind})\{\bar{A}\} \quad \Rightarrow \quad \Delta \vdash \text{Ind}(X: \text{Kind})\{\bar{A}\}
\]
\[
A = \text{Ctor}(i, I) \quad \Rightarrow \quad I = \text{Ind}(X: \text{Kind})\{\bar{A}\}
\]
\[
A = \text{Elim}[X,I]\{A\}\{\bar{B}\} \quad \Rightarrow \quad I = \text{Ind}(X: \text{Kind})\{\bar{A}\}
\]
\[
\Delta \vdash A : I
\]
\[
\Delta \vdash A' : \text{Kscm}
\]
\[
\Delta \vdash B : \text{Kscm} \text{ and } B = \beta \eta i A'
\]
\[
\Delta \vdash B_i : \Psi X, i(A_i, A')
\]

\[
\Delta \vdash A_i : I
\]
\[
\Delta \vdash A' : \text{Kscm}
\]
\[
\Delta \vdash B : \text{Kscm} \text{ and } B = \beta \eta i A'
\]
\[
\Delta \vdash B_i : \Psi X, i(A_i, A')
\]

\[
\Delta \triangleright_{\beta \iota} \Delta'
\]

\[
\Delta \vdash A' : B
\]
\[
\Delta' \vdash A : B
\]

**Proof**  The interesting cases are the \text{APP} and \text{ELIM}.

- \text{APP} When only the sub-terms reduce without a reduction at the head, the lemma follows by using the induction hypothesis on the sub-terms. Suppose that

\[
\Delta \vdash \lambda X : A_1. A_2 \vdash \Pi X : B'. A' \quad \Delta \vdash B : B'
\]

and \(A \; \beta \iota \; [B/X]A_2\). We know from lemma 19 that

\[
\Delta, X : A_1 \vdash A_2 : A_3
\]

\[
\Pi X : A_1. A_3 = \beta \eta i \Pi X : B', A'
\]

\[
\Delta \vdash A_1 : s_1
\]

This implies that \(A_1 = \beta \eta i B'\) and \(A_3 = \beta \eta i A'\). Moreover,

\[
\text{Cls}(B') = \text{Cls}(A_1) = \text{lift}(\text{Cls}(X))
\]

Therefore, we get from lemma 14 that

\[
\text{Cls}(s_2) = \text{Cls}(s_1) \Rightarrow s_2 = s_1
\]

Applying the \text{CONV} rule we get that \(\Delta \vdash B : A_1\). By lemma 15 we get that \(\Delta \vdash [B/X]A_2 : [B/X]A_3\). We can show in a similar manner as before that \(\text{Cls}(A_3) = \text{Cls}(A')\). This allows us to apply the \text{CONV} rule again which leads to the required result.

- \text{L-ELIM} We will only consider the case when an \(\epsilon\) reduction takes place at the head. The other cases follow easily by structural induction.

\[
\Delta \vdash A : I
\]
\[
\Delta \vdash A' : \text{Kscm}
\]

for all \(i\)

\[
\Delta \vdash \text{Elim}[X,I]\{A\}\{\bar{B}\} : A'
\]

where \(I = \text{Ind}(X: \text{Kind})\{\bar{C}\}\) and \(\forall i. \text{small}(C_i)\).

The interesting case is when we consider the reduction

\[
\text{Elim}[X,I]\{\bar{A}\}\{\text{Ctor}(i, I)\}
\]

Suppose \(A'' = \text{Elim}[X,I]\{\text{Ctor}(i, I)\} A\). Suppose that \(A\) and \(A''\) are different cases by which \(X\) is a kind of a constructor.

- If \(C_i = X\) and \(A'' = \text{B}i\). From definition 9 we can see that in this case, \(B_i\) has the type \(A''\).
- If \(C_i = \Pi Y : B. C\), then \(A'' = \text{Ind}(X, Y)\{\bar{C}_i\} A_i\). We have that \(\Delta \vdash B_i : \Psi X, i(C_i, A_i)\). By the induction hypothesis, the redcut has type \(A''\).
- If \(C_i = \Pi Y : B. X \rightarrow C\), then \(A'' = \Phi X, i.Bi. C_i, A_i (\Psi X, i. B_i)\). From Definition 9 we have that \(\Delta \vdash \Psi X, i. B_i \rightarrow \{[X/Y]\} A_i \rightarrow \Psi X, i. (C_i, A_i)\). We also know that \(\Delta \vdash A_i : [X/Y] A_i\). From here, we can apply the induction hypothesis and show that the redcut has type \(A''\).

**Lemma 21 (Subject reduction for \(\beta \iota\) reduction)** If the judgment \(\Delta \vdash A : B\) is derivable, and if \(A \triangleright_{\beta \iota} A'\) and

\[
A \triangleright_{\beta \iota} A'
\]

then we have that

\[
\Delta \vdash A' : B
\]

\[
\Delta' \vdash A : B
\]

**Corollary 22** Suppose \(A\) is a well typed term. If \(A \triangleright_{\beta \iota} A'\), then \(A \triangleright_{\beta \iota} A'\).

**D.2.4 Reductions on well typed terms**

**Lemma 21** Suppose \(A\) is a well typed term. If \(A \triangleright_{\beta \iota} A'\), then \(A \triangleright_{\beta \iota} A'\).
• **ELIM** We will only consider the case when an ε reduction takes place at the head. The other cases follow easily by structural induction.

\[
\Delta \vdash A : I \quad \Delta \vdash A' : I \rightarrow \text{Kind}
\]

For all \( i \)

\[
\Delta \vdash B_i : \zeta_{\Pi\lambda\eta}(C_i, A', \text{Ctor}(i, I))
\]

\[
\Delta \vdash \text{Elim}[I, A']((A)(\vec{B})) : A' A^i
\]

where \( I = \text{Ind}(X : \text{Kind})\{\vec{C}\} \)

The interesting case is when we consider the reduction

\[
\text{Elim}[I, A']((\text{Ctor}(i, I) A)(\vec{B})) \vdash_1 (\Phi_{X, I, B'}(C_i, B_i)) \bar{A}
\]

\[
\text{where } I = \text{Ind}(X : \text{Kind})\{\vec{C}\}
\]

\[
B^i = \lambda Y : I_1. (\text{Elim}[I_1, A'_i](Y)(\vec{B}))
\]

Suppose \( A'' = (\Phi_{X, I, B'}(C_i, B_i)) \bar{A} \). Suppose that \( \bar{A} = A_{1...n} \). We have that \( \Delta \vdash B_i : \zeta_{\Pi\lambda\eta}(C_i, A', \text{Ctor}(i, I)) \). By using the inversion lemma we can get that \( \Delta \vdash B' : \Pi X : I, A' X \). By induction on the structure of \( C_i \) (where \( C_i \) is a kind of a constructor), we can show that if \( C_i = \Pi Y : B, X \), then \( \Delta \vdash \Phi_{X, I, B'}(C_i, B_i) : \Pi Y : B, A' \text{Ctor}(i, I) Y \). The required result follows from here.

\[\square\]

**Corollary 23** Suppose \( A \) is a well-formed term. If \( A \vdash_\eta A' \), then \( A \vdash_\eta A' \) and \( A' \) is well formed.

**Corollary 24** Suppose \( A \) is a well-formed term. If \( A \vdash_\eta A' \), then there exists a well-formed term \( A'' \) such that \( A \vdash_\eta A'' \) and \( A'' \) is well-formed.

**Lemma 22** Let \( \Delta \vdash A : B \) and \( \Delta \vdash A' : B' \) be two derivable judgments. If \( A =_\beta_a A' \), then \( \text{Cls}(A) = \text{Cls}(A') \).

**Proof** We know that \( \| A \| \) and \( \| A' \| \) have a common reduct, say \( A_2 \). This implies that

\[
\| A \| \vdash_\beta_{\eta} B \vdash_\eta A_2 \quad \text{and} \quad \| A' \| \vdash_\beta_{\eta} B' \vdash_\eta A_2
\]

From here we get that

\[
A \vdash_\eta B_0 \quad \text{and} \quad A' \vdash_\eta B'_0 \quad \text{where} \quad \| B_0 \| = B \quad \text{and} \quad \| B'_0 \| = B'
\]

Beta reduction does not change the class of a term. Moving from marked to unmarked terms also does not change the class of a term. Therefore, we get that

\[
\text{Cls}(A) = \text{Cls}(B_0) = \text{Cls}(B) = \text{Cls}(A_2) \quad \text{and} \quad \text{Cls}(A_2) = \text{Cls}(B') = \text{Cls}(B'_0) = \text{Cls}(A')
\]

\[\square\]

**Corollary 25** Let \( \Delta \vdash A : s_1 \) and \( \Delta \vdash B : s_2 \) be two derivable judgments. If \( A =_\beta_a B \), then \( s_1 = s_2 \).

**Lemma 23** If \( \Delta_1, Y : C, \Delta_2 \vdash A : B \) and \( Y \notin \text{FV}(\Delta_2) \cup \text{FV}(A) \), then there exists a \( B' \) such that \( \Delta_1 \Delta_2 \vdash A : B' \). (This also implies that \( B =_\beta_a B' \).

**Proof** The proof is by induction on the structure of the derivation. We will consider only the important cases.

• **case FUN.** We know that

\[
\Delta_1, Y : C, \Delta_2, X : A \vdash B : B' \\
\Delta_1, Y : C, \Delta_2 \vdash \Pi X : A, B' : s \\
\Delta_1, Y : C, \Delta_2 \vdash \lambda X : A, B : \Pi X : A, B'
\]

Applying the induction hypothesis to the formation of \( B \)

\[
\Delta_1 \Delta_2, X : A \vdash B : C' \\
B' =_\beta_{\eta} C'
\]

By lemma 18 we have that

\[
\Delta_1 \Delta_2, X : A \vdash B : C' \quad \text{which implies} \quad \Delta_1 \Delta_2 \vdash \Pi X : A, C' : s
\]

Therefore we get that

\[
\Delta_1 \Delta_2 \vdash \lambda X : A, B : \Pi X : A, C'
\]

• **case APP** We know that

\[
\Delta_1, Y : C, \Delta_2 \vdash A : \Pi X : B', A' \\
\Delta_1, Y : C, \Delta_2 \vdash B : B'
\]

\[
\Delta_1, Y : C, \Delta_2 \vdash A : [B/X]A'
\]

By applying the induction hypothesis we get that

\[
\Delta_1 \Delta_2 \vdash A : A_2 \quad \text{and} \quad \Delta_1 \Delta_2 \vdash B : A_3 \quad \text{where} \quad A_2 =_\beta_{\eta} \Pi X : B', A' \quad \text{and} \quad A_3 =_\beta_{\eta} B'
\]

From lemma 13, \( A_2 \vdash_\beta a \bar{A} : \bar{A} : (\Pi X : B'', A'') B'. \) Since \( \beta_a \) reduction preserves type, and \( A_2 \) is well formed, we have that \( A_2 \vdash_\beta a \bar{A} : \Pi X : B'', A'' \). This implies that \( A''' =_\beta_{\eta} A' \) and \( B''' =_\beta_{\eta} B' \). We also get that \( A_3 =_\beta_{\eta} B''' \). From corollary 25 we get that \( A_3 \) and \( B''' \) have the same sort. By applying the \( \text{CONV} \) rule we get that

\[
\Delta_1 \Delta_2 \vdash A : \Pi X : B'', A'' \quad \text{and} \quad \Delta_1 \Delta_2 \vdash B : B''
\]

Therefore, we get that

\[
\Delta_1 \Delta_2 \vdash A : [B/X]A''
\]

As a corollary we now get that

**Lemma 24 (Strengthening)** If \( \Delta_1, Y : C, \Delta_2 \vdash A : B \) and \( Y \notin \text{FV}(\Delta_2) \cup \text{FV}(A) \), then \( \Delta_1 \Delta_2 \vdash A : B \).

**Lemma 25 (Subject reduction for \( \eta \) reduction)** If \( \Delta \vdash A : B \), and \( A \vdash_\eta A' \) and \( \Delta \vdash_\eta A' \), then we have that

\[
\Delta \vdash A' : B \quad \Delta' \vdash A : B
\]

**Proof** The interesting case is that of functions. Suppose that \( \Delta \vdash \lambda X : A_1, A_2 X : B \) \( X \notin \text{FV}(A_2) \) \( \lambda X : A_1, A_2 X \vdash_\eta A_2 \) From lemma 19 we know that

\[
\Delta, X : A_1 \vdash A_2 X : A_3 \quad B =_\beta_{\eta} \Pi X : A_1, A_3 \quad \Delta \vdash B : s
\]

Again applying lemma 19 we get that

\[
\Delta, X : A_1 \vdash A_2 \Pi Y : B', A' B' =_\beta_{\eta} A_1 \quad A_3 =_\beta_{\eta} [X/Y]A'
\]

By applying the \( \text{CONV} \) rule now, we get that \( \Delta, X : A_1 \vdash A_2 : B \). By applying lemma 24 we get that \( \Delta \vdash A_2 : B \).

**Theorem 26 (Subject reduction)** If \( \Delta \vdash A : B \), and \( A \vdash_\eta A' \) and \( \Delta \vdash_\eta A' \), then we have that

\[
\Delta \vdash A' : B \quad \text{and} \quad \Delta' \vdash A : B
\]

**Proof** Follows from lemma 21 and 25. \[\square\]
D.3 Strong Normalization

The proof is structured as follows:

- We introduce a calculus of pure terms. This is just the pure \( \lambda \) calculus extended with a recursive filtering operator. We do this so that we can operate in a confluent calculus.
- We define a notion of reducibility candidates. Every schema gives rise to a reducibility candidate. We also show how these candidates can be constructed inductively.
- We then 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 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.

We then reduce the strong normalization of all well formed terms to the strong normalization of pure terms.

D.3.1 Notation

The syntax for the language is:

\[
\begin{align*}
\langle \text{ctx} \rangle &::= \cdot | \Delta, X : A \\
\langle \text{sort} \rangle &::= \text{Kind} | \text{Kscm} | \text{Ext} \\
\langle \text{var} \rangle &::= z | k | t \\
\langle \text{ptm} \rangle &::= s | X | \lambda X : A. B | A B | \Pi X : A. B \\
&| \text{Ind}(X : \text{Kind})\{A\} | \text{Ctor}(i, A) \\
&| \text{Elim}[A', B'](A)\{B\}
\end{align*}
\]

The proof of strong normalization uses the stratification in the language shown below.

\[
\begin{align*}
\langle \text{ctx} \rangle &::= \cdot | \Delta, z : \text{Kscm} | \Delta, k : u | \Delta, t : \kappa \\
\langle \text{kscm} \rangle &::= z | \Pi: \kappa. u | \Pi u. u_1. u_2 | \text{Kind} \\
\langle \text{kind} \rangle &::= k | \Pi: \kappa_1. \kappa_2 | \kappa[\tau] | \lambda k : u. \kappa | \kappa_1 \kappa_2 \\
&| \Pi u. \kappa | \Pi: \text{Kscm}. \kappa \\
&| \text{Ind}(k : \text{Kind})\{\vec{k}\} | \text{Elim}[\kappa', u](\tau)\{\vec{k}\}
\end{align*}
\]

\[
\begin{align*}
\langle \text{type} \rangle &::= t | \lambda t : \kappa. \tau | \tau t : \kappa | \lambda k : \text{Kscm}. \tau | \kappa[t] \\
&| \lambda z : \text{Kscm}. \tau | \tau[k] | \text{Ctor}(i, \kappa) \\
&| \text{Elim}[\kappa, \kappa](\tau')(\tau)\{\vec{r}\} | \text{Elim}[\kappa, \kappa](\tau')(\tau')\{\vec{r}\}
\end{align*}
\]

In this section, the types are also referred to as proof terms. We sometimes use \( I \) to refer to an inductive definition.

D.3.2 Pure terms

The pure terms are defined as:

\[
(\lambda t. a) \ b \triangleright_{\beta} [b/t] a
\]

The set of reductions on the pure terms are defined as:

\[
\lambda t. (a t) \triangleright_{\eta} a \quad \text{if } t \notin FV(a)
\]

match \( t.\{\vec{a}\} \) \( (\text{Co}(i) \ b) \triangleright_{\text{Ind}} \text{match } t.\{\vec{a}\}/t[\alpha_i] \ b \)

The translation from types to pure terms is defined as:

\[
|t| = t \\
|\tau_1 \tau_2| = \tau_1 | \tau_2 \\
|\tau| \kappa = \tau \\
|\tau u| = \tau \\
|\lambda t : \kappa. \tau| = \tau \\
|\lambda k : u. \tau| = \tau \\
|\lambda z : \text{Kscm}. \tau| = \tau \\
|\text{Ctor}(n, \kappa)| = \text{Co}(n) \\
|\text{Elim}[\kappa, \kappa](\tau)\{\vec{r}\}| = \text{match } t.\{\vec{Y}(\kappa_1, \tau_1, \lambda \kappa_2.t \tau_2)\} | \tau| \\
\]

where \( \kappa = \text{Ind}(k : \text{Kind})\{\vec{r}\} \) and

\[
\mathcal{T}(\kappa, \alpha_1, \alpha_2) = a_1 \\
\mathcal{T}(\Pi \kappa : \kappa_1. \kappa_2, \alpha_1) = \lambda t. \mathcal{T}(\kappa_2, \alpha_1, t, \alpha_2) \\
\mathcal{T}(\Pi u : \kappa. \kappa, \alpha_1) = \mathcal{T}(\kappa, \alpha_1, t, \alpha_2) \\
\mathcal{T}(\Pi z : \text{Kscm}. \kappa, \alpha_1) = \mathcal{T}(\kappa, \alpha_1, t_1, \alpha_2) \\
\mathcal{T}(\Pi \vec{X} : A, k \rightarrow \kappa, \alpha_1) = \lambda t. \mathcal{T}(\kappa, \alpha_1, t \mid \vec{X}, \alpha_2)\{t \mid \vec{X}\}(\alpha_2)
\]

Lemma 26 Let \( \tau \) and \( \tau' \) be two well formed types and let \( \tau \) be a type variable. Then \( |\tau'/\tau| = |\tau'/\tau| | \tau| \).

Proof It is a straightforward proof by induction over the structure of \( \tau \).

The following lemma uses Definitions 9 and 7 in Section D.2 and also the definition of \( \mathcal{T} \) from above.

Lemma 27 \[ |\Phi_{X, t, b}(\kappa, \tau)| = |\mathcal{T}(\{\vec{Y}(\kappa_1, \tau_1, \lambda \kappa_2.t \tau_2)\}/t)\mathcal{T}(\kappa, \tau_1, \lambda \kappa_2.t \tau_2)\].

Proof The proof is by induction on the fact that \( \kappa \) is the kind of a constructor.

Lemma 28 For all well formed proof terms \( \tau_1 \) and \( \tau_2 \), if \( \tau_1 \triangleright_{\beta} \tau_2 \), then \( |\tau_1| \triangleright_{\beta} |\tau_2| \) where \( j \leq i \).

Proof Follows from lemmas 26 and 27.

D.3.3 Interpretation of schemas

Definition 27 (Arity) We call ground kind schemas arities denoted as \( \text{arity}(u, \text{Kind}) \). The arities are defined with the following grammar:

\[
\langle \text{kscm} \rangle u ::= \text{Kind} | \Pi u. u_1. u_2 | \Pi u. \kappa. u
\]

Definition 28 (Schema map) We define a kind schema mapping \( \mathcal{K} \) as a function mapping kind schema variables \( z \) to arities. We also use \( \mathcal{K}, z : u \) to say that \( \mathcal{K} \) has been augmented with the mapping \( z \mapsto u \).
Definition 29 We define the function \( \rho(u)_K \) as:
\[
\rho(u)_K = \rho_0(K(u)) \quad \text{where}
\]
- \( \rho_0(\text{Kind}) \) is the set of sets of pure terms;
- \( \rho_0(\Pi : \kappa : u_1, u_2) \) is the set of functions from \( \rho_0(u_1) \) to \( \rho_0(u_2) \); and
- \( \rho_0(\Pi t : \kappa : u) \) is the set of functions from \( \Lambda \) to \( \rho_0(u) \).

Definition 30 For each kind schema \( u \) and mapping \( K \), we define in \( \rho(u)_K \) the relation of partial equivalence written as \( \simeq_{K(u)} \) as follows:
- for all \( C \) and \( C' \) in \( \rho_0(\text{Kind}) \), we have that \( C \simeq_{\text{Kind}} C' \iff C = C' \);
- for all \( C \) and \( C' \) in \( \rho_0(\Pi : u_1, u_2) \), we have \( C \simeq_{\Pi u_1, u_2} C' \iff \text{for all } C_1 \text{ and } C_2 \text{ in } \rho_0(u_1) \text{ with } C_1 \simeq_{u_1} C_2 \text{ we get that } C_1 \simeq_{u_2} C'_1 \); and
- for all \( C \) and \( C' \) in \( \rho_0(\Pi : \kappa : u) \), we have \( C \simeq_{\Pi \kappa : u} C' \iff \text{for all } a \text{ and } b \text{ in } \Lambda \text{ such that } a =_{\rho_0} b, \text{ we get that } C a \simeq_u C' b \).

Definition 31 (Invariant) Given \( C \) in \( \rho(u)_K \), we say that \( C \) is invariant \( \iff C \simeq_{K(u)} C \).

Definition 32 (Neutral terms) A term is called neutral if it has neither of the following forms – \( \lambda t. a, \text{Co}(i) \bar{a}, \text{or } \text{match } t\{\bar{a}\} \).

Definition 33 We define \( \mathcal{CR}_0(\text{Kind}) \) as consisting of all sets \( C \) such that:
- if \( a \in C \), then \( a \) is strongly normalizing;
- if \( a_1 \triangleright a_2 \) and \( a_1 \in C \), then \( a_2 \in C \); and
- if \( a \) is neutral and for all terms \( a' \) such that \( a \triangleright a' \) and \( a' \in C \), then \( a \in C \).

Definition 34 (Candidates) We define \( \mathcal{CR}(u)_K \) as a subset of \( \rho(u)_K \) as:
\[
\mathcal{CR}(u)_K = \mathcal{CR}_0(K(u)) \quad \text{where}
\]
- \( \mathcal{CR}_0(\text{Kind}) \) is defined as in Definition 33;
- \( \mathcal{CR}_0(\Pi : \kappa : u) \) is the set of invariant elements \( C \) belonging to \( \rho_0(\Pi : \kappa : u) \) such that \( C \Lambda \subseteq \mathcal{CR}_0(u) \); and
- \( \mathcal{CR}_0(\Pi \kappa : u_1, u_2) \) is the set of invariant elements \( C \) belonging to \( \rho_0(\Pi \kappa : u_1, u_2) \) such that \( C \subseteq \mathcal{CR}_0(u_1) \subseteq \mathcal{CR}_0(u_2) \).

Proposition 35 All reducibility candidates are invariant.

Proposition 36 Let \( (C_i)_{i \in I} \) be a family of reducibility candidates of Kind indexed by a set \( I \). Then \( \bigcap_{i \in I} C_i \) is a reducibility candidate of schema Kind.

Lemma 29 Let \( C \in \rho(u)_K \). If \( C \) is invariant, then
\[
C \in \mathcal{CR}(u)_K \iff \forall C' \in \text{Dom}(\mathcal{CR}(u)_K) . C C' \in \mathcal{CR}(\text{Kind}_K)
\]

Proof Straightforward induction over the structure of \( K(u) \). \( \square \)

Definition 37 Let \( a_1 \) be a strongly normalizing term. Then the length of the longest sequence of reductions to a normal form is denoted as \( \nu(a_1) \).

Lemma 30 Let \( a_1 \) and \( a_2 \) be two terms and let \( C \in \mathcal{CR}_0(\text{Kind}) \) be a reducibility candidate. If \( a_2 \) is strongly normalizing, and if \( [a_2/t]a_1 \in C \), then \( (\lambda t. a_1) a_2 \in C \).

Proof By induction over \( \nu(a_1) + \nu(a_2) \). \( \square \)

Corollary 38 Let \( a_1 \) be a pure term and let \( C \) be a reducibility candidate of schema Kind. Let \( t \) and \( \bar{a}' \) be respectively a sequence of variables and terms of the same length. If for all \( i \), \( a'_i \) is strongly normalizing, and if \( [a'_i/t]a_1 \in C \), then \( (\lambda t a_1) \bar{a}' \in C \).

Lemma 31 For all reducibility candidates \( C \) of kind Kind, for all sequences of strongly normalizing \( \bar{a} \) and \( \bar{b} \) and for all \( i \) less than the length of \( \bar{a} \), we have that
\[
\text{match } t\{\bar{a}\} \bar{b} \in C \iff [\text{match } t\{\bar{a}\}/t[a_1] \bar{b} \in C
\]

Proof Follows by induction over \( \nu(a_1) + \nu(b_i) \) (for all \( i \)). \( \square \)

Definition 39 (Canonical candidates) Define \( \text{Can}(u)_K \) as:
\[
\text{Can}(u)_K = \text{Can}_0(K(u)) \quad \text{where}
\]
- \( \text{Can}_0(\text{Kind}) \) is the set of all strongly normalizing terms;
- \( \text{Can}_0(\Pi : \kappa : u) \) is the function mapping all pure terms to \( \text{Can}_0(u) \); and
- \( \text{Can}_0(\Pi \kappa : u_1, u_2) \) is the function mapping all elements of \( \rho_0(u_1) \) to \( \text{Can}_0(u_2) \).

D.3.4 Properties of candidates

In this section, we state some properties of the reducibility candidates. The properties with respect to the union and the intersection of a family of candidates will be used for the inductive constructions of candidates.

Definition 40 (Order over candidates) For each kind schema \( u \) and mapping \( K \), we define in \( \rho(u)_K \) the relation \( <_{K(u)} \) as follows:
- for all \( C \) and \( C' \) in \( \rho_0(\text{Kind}) \), we have that \( C <_{\text{Kind}} C' \iff C \subseteq C' \);
- for all \( C \) and \( C' \) in \( \rho_0(\Pi : \kappa : u_1, u_2) \), we have \( C <_{\Pi u_1, u_2} C' \iff \text{for all } C_1 \text{ in } \rho_0(u_1), \text{ we get that } C_1 \subseteq C' \);
- for all \( C \) and \( C' \) in \( \rho_0(\Pi : \kappa : u) \), we have \( C <_{\Pi \kappa : u} C' \iff \text{for all } a \text{ in } \Lambda, \text{ we get that } C a <_{\kappa} C' a \).

Definition 41 For all schemas \( u \) and mapping \( K \), for all families of elements in \( \rho(u)_K \), we define \( \bigwedge_{i \in I} C_i \) as:
- for all \( C_i \in \rho_0(\text{Kind}) \), \( \bigwedge_{i \in I} C_i = \bigcap_{i \in I} C_i \);
- for all \( C_i \in \rho_0(\Pi : \kappa : u) \), \( \bigwedge_{i \in I} C_i = b = \in \Lambda \iff \bigwedge_{i \in I} C_i \); and
- for all \( C_i \in \rho_0(\Pi \kappa : u_1, u_2) \), \( \bigwedge_{i \in I} C_i = C^* \in \rho_0(u_1) \iff \bigwedge_{i \in I} C_i \bigwedge_{i \in I} C^* \).

Lemma 32 Let \( u \) be a schema and \( K \) a mapping and \( C_i \) a family of elements of \( \rho(u)_K \). Then \( \forall j \in I, \bigwedge_{i \in I} C_i <_{K(u)} C_j \).

Proof It follows in a straightforward way by induction over the structure of \( K(u) \). \( \square \)

The following two propositions also follow easily by induction over the structure of \( K(u) \).
Proposition 42 Let $u$ be a schema and $K$ a mapping and $C_i$ a family of elements of $\rho(u)K$. If all $C_i$ are invariants, then the same holds for $\bigwedge_{i \in I} C_i$.

Proposition 43 Let $u$ be a schema and $K$ a mapping and $C_i$ a family of elements of $CR(u)K$. Then we also have that $\bigwedge_{i \in I} C_i \in CR(u)K$.

Corollary 44 We get that $(CR(u)K, <_{K(u)})$ is an inf-semi-lattice for all schema $u$ and mapping $K$. We use $\text{min}(K(u))$ to denote the smallest element.

Definition 45 For all schemas $u$ and mapping $K$, for all families of elements in $\rho(u)K$, we define $\bigvee_{i \in I} C_i$ as:
- for all $C_i \in \rho_0(\text{Kind})$, $\bigvee_{i \in I} C_i = \bigcup_{i \in I} C_i$;
- for all $C_i \in \rho_0(\Pi \kappa : \kappa, u)$, $\bigvee_{i \in I} C_i = \{ b \in \Lambda \mapsto \bigvee_{i \in I} C_i \}$; and
- for all $C_i \in \rho_0(\Pi k : u_1, u_2)$, $\bigvee_{i \in I} C_i = C' \in \rho_0(u_1) \mapsto \bigvee_{i \in I} C_i \mapsto C'$.

Lemma 33 Let $u$ be a schema and $K$ be a mapping. Let $(C_i)_{i \in I}$ and $(C'_i)_{i \in I}$ be two families of elements of $\rho(u)K$. If for all elements $i$ of $I$ we have that $C_i \succeq_{K(u)} C'_i$, then we also have that $\bigvee_{i \in I} C_i \succeq_{K(u)} \bigvee_{i \in I} C'_i$.

Proof Straightforward induction over the structure of $K(u)$.

Corollary 46 Let $u$ be a schema and $K$ be a mapping. Let $(C_i)_{i \in I}$ be a family of elements of $\rho(u)K$. If all $C_i$ are invariant, then $\bigvee_{i \in I} C_i$ is also invariant.

Lemma 34 Let $u$ be a schema and $K$ be a mapping. Let $(C_i)_{i \in I}$ be a family of elements of $\rho(u)K$ and $C \in \rho(u)K$. If for all $i$, $C_i \succeq_{K(u)} C$, then $\bigvee_{i \in I} C_i \succeq_{K(u)} C$.

Proof The proof is by induction over the structure of $K(u)$.

Lemma 35 Let $(C_i)_{i \in I}$ be a totally ordered family of elements of $CR(u)K$. Then $\bigvee_{i \in I} C_i \in CR(u)K$.

Proof The proof is by induction over the structure of $K(u)$. Suppose $\bigvee_{i \in I} C_i = C'$. Then $K(u) = \text{Kind}$. We have to make sure that all three conditions in Definition 33 are satisfied. The first two conditions follow obviously. For the third case, assume that $a$ is neutral and for all terms $a_i$ such that $a \triangleright a_i$, we have that $a_i \in C'$. This implies that $a_i \notin C_j$ for some $j$. Since there are finitely many such $C_j$ and they are totally ordered, we can choose a $C_k$ among them that contains all the $C_j$s. Since this $C_k$ is also a candidate, it contains $a$. Therefore, $a \in \bigvee_{i \in I} C_i$.

$K(u) = \Pi \kappa : \kappa, u$. Since all the $C_i$ are invariant, it follows from Definitions 30 and 31 that for a term $a \in \Lambda$, we have that $C_i \in \Lambda$ is invariant. Again from Definition 40, it is clear that the $C_i \in \Lambda$ are totally ordered. Also from Corollary 46 we get that $\bigvee_{i \in I} C_i \in \rho_0(u)K$. From Definition 34, it follows that $\bigvee_{i \in I} C_i \in CR_0(\Pi \kappa : \kappa, u)$.

$K(u) = \Pi k : u_1, u_2$. Similar to the previous case.

Definition 47 (Schema interpretation) A schema interpretation $U$ is a function that maps a kind variable $k$ to an element of $\rho(u)K$. We also use $U, k : C$ to say that $U$ has been augmented with the mapping $k \mapsto C$.

Definition 48 (Well formed kinds) Let $u$ be a schema, $\kappa$ be a kind, $K$ be a mapping, and $U$ be an interpretation. We say that $\kappa$ is a well formed kind of schema $K(u)$ under $K$ and $U$ iff:

1. $\kappa = k$ and $U(k) = \rho(u)K$;
2. $\kappa = \Pi \kappa_1, \kappa_2$ with $K(u) = \beta_{\pi_1}$. Kind and $\kappa_1$ and $\kappa_2$ are both well constructed of schema Kind under $K$ and $U$;
3. $\kappa = \Pi k : u', \kappa'$ with $K(u) = \beta_{\pi_1}$. Kind and $\kappa'$ is well constructed of schema Kind under $K$ and $U, k : \rho(u)'K$;
4. $\kappa = \Pi z : KSCM, \kappa'$ with $K(u) = \beta_{\pi_1}$. Kind and for all $u'$ such that $u' \in \text{arity}(u_1, \text{Kind})$, we have that $\kappa'$ is well constructed of schema Kind under $K, z : u'$ and $U$;
5. $\kappa = \kappa_1 \kappa_2$ if there exists two schemas $u_1$ and $u_2$ with $\kappa_2$ well constructed of schema $K(u)u_2$ under $K$ and $U$, also $\kappa_1$ well constructed of schema $K(\Pi k : u_1, u_2, u_1)$ under $K$ and $U$, and $\rho(u)K = \rho(\kappa_1 / k, u_1, u_2)K$;
6. $\kappa = \kappa_1 \tau_1$ if there exists a schema $u_2$ and kind $\kappa_2$ such that $\kappa_2$ is well constructed of schema $K(\Pi \kappa : \kappa_2, u_2)$ under $K$ and $U$ and $\rho(u)K = \rho(\kappa_1 / k, u_2)K$;
7. $\kappa = \lambda k : u_1, \kappa_1$ if there exists a $u_2$ such that $\kappa_1$ is well constructed of schema $K(u_2)$ under $K$ and $U, k : \rho_0(\text{Kind})$ and $\rho(u)K = \rho(\Pi k : u_1, u_2)K$;
8. $\kappa = \lambda \tau : \kappa_1, \kappa_2$ if there exists a $u_2$ such that $\kappa_2$ is well constructed of schema $K(u_2)$ under $K$ and $U$ and $\rho(u)K = \rho(\Pi \kappa : \kappa_2, u_2)K$;
9. $\kappa = \text{Ind}(k : \text{Kind})K \kappa'$ if all $\kappa_i$ are kinds of constructors and well constructed of schema Kind under $K$ and $U$, $k : \rho_0(\text{Kind})$, and $\rho(u)K = \rho(\Pi k : \kappa_1, u_2)K$;
10. $\kappa = \text{Elim}[k', u'](\kappa)[\kappa']$ if $\kappa' = \text{Ind}(k : \text{Kind})K \kappa'$, and $\kappa'$ is well constructed of schema Kind under $K$ and $U$, also $u'$ is a schema and $K(u) = \beta_{\pi_1} u'$, and $\kappa_i$ is well constructed of schema Kind under $K, \kappa'_{u'}K$ under $K$ and $U$. $\square$

Definition 49 We define compatible mappings and interpretation as:

1. A mapping $K$ is compatible with a context $\Delta$ if for all $z \in \Delta$, we have $K(z) = \text{arity}(u, \text{Kind})$.
2. An interpretation $U$ is compatible with a context $\Delta$ and a compatible mapping $K$ if for all pairs $(k, u) \in \Delta$, we have $U(k) \in \rho(u)K$.

Lemma 36 If $\Delta \vdash k : u$, then for all compatible $K$ and $U$, we have that $\kappa$ is well constructed of schema Kind.

Proof By induction over the structure of $\kappa$. $\square$
D.3.5 Inductive constructions

Consider an increasing function \( F \) in \( \rho_0(\text{Kind}) \) for the order \( \prec_{\text{Kind}} \). Denote the smallest element of \( \rho_0(\text{Kind}) \) as \( \bot \). Since \( \rho_0(\text{Kind}) \) is closed under \( \cap \), and \( \rho(\text{Kind}), \prec_{\text{Kind}} \) is an inf-semi-lattice, the function \( F \) has a least fixed point \( (\mathcal{I}F) \). We will construct this least fixed point inductively. We first define the transfinite iteration of \( F \).

**Definition 50** Let \( C \in \rho_0(\text{Kind}) \) and \( o \) be an ordinal. We define the iteration of order \( o \) of \( F \) over \( C \) as:
- \( F^0(C) = C \);
- \( F^{o+1}(C) = F(F^o(C)) \); and
- \( F^{\text{lim}(U)} = \bigcup_{o \in U} F^o(C) \).

**Lemma 37** Let \( o \) be an ordinal; we have \( F^o(\bot) \prec_{\text{Kind}} \mathcal{I}Fp(F) \).

**Proof** The proof is by induction over \( o \). If \( o = 0 \), then it follows immediately. Otherwise,
- \( o = o' + 1 \) Then we have that \( F^o(\bot) = F(F^{o'}(\bot)) \).
  By the induction hypothesis, we get that \( F(F^{o'}(\bot)) \prec_{\text{Kind}} \mathcal{I}Fp(F) \). This implies that \( F(F^{o'}(\bot)) \prec_{\text{Kind}} \mathcal{I}Fp(F) \).
- \( o = \text{lim}(U) \) Follows immediately from the induction hypothesis and lemma 34.

**Remark 51** Since we do not consider the degenerate case of \( F(\bot) = \bot \), it follows from lemma 37 that for some ordinal \( o \), we have that \( \mathcal{I}Fp(F) = F^o(\bot) \).

**Lemma 38** Suppose \( S \) is a subset of \( \rho_0(\text{Kind}) \) satisfying:
- \( (C_i)_{i \in I} \) is a totally ordered family of elements of \( S \), then \( \bigcup_{i \in I} C_i \in S \);
- \( F(\bot) \in S \); and
- for all \( C \in S \), \( F(C) \in S \).

Then \( \mathcal{I}Fp(F) \in S \).

**Proof** Follows from the fact that \( \mathcal{I}Fp(F) = F^o(\bot) \) for some ordinal \( o \).

**Definition 52** Let \( a \in \mathcal{I}Fp(F) \). We define \( \deg(a) \) as the smallest ordinal such that \( a \in F^{\deg(a)}(\bot) \).

**Definition 53** To all \( a \in \mathcal{I}Fp(F) \), we associate \( \text{pred}(a) \) defined as \( F^{\deg(a)-1}(\bot) \).

**Lemma 39** For all \( a \), \( \deg(a) \) is an ordinal successor.

**Proof** Suppose it is the limit of the set \( U \). From Definition 50, there exists some \( o \in U \) for which \( a \in F^o(\bot) \). This leads to a contradiction.

**Definition 54 (Partial order)** Suppose \( C \) and \( C' \) are two elements of \( \mathcal{C}R_o(\text{Kind}) \). We say that \( C \preceq F C' \) if \( C = F^o(\bot) \) and \( C' = F^{o'}(\bot) \), and \( o < o' \).

D.3.6 Interpretation of kinds

In this section we interpret kinds as members of reducibility candidates. First we augment the schema interpretation

**Definition 55** We augment \( \mathcal{U} \) so that it maps a kind variable to an element of \( \rho(u)\kappa \), and a type variable to a pure term \( a \).

**Definition 56** We denote the interpretation of a type \( \tau \) as \( C^F_{\mathcal{U}}(\tau) \).
To form this, we first construct the corresponding pure term \( |\tau| \) and then substitute the type variables by the corresponding pure terms in \( \mathcal{U} \). This is equivalent to \( \mathcal{U}(|\tau|) \).

**Definition 57 (Interpreting kinds)** Consider a kind \( \kappa \), a schema \( u \), a mapping \( K \), and an interpretation \( \mathcal{U} \). Suppose \( \kappa \) is well constructed of schema \( K(u) \) under \( K \) and \( \mathcal{U} \). We define by recursion on \( \kappa \):

1. \( C^F_{\mathcal{U}}(k) = U(k) \)
2. \( C^F_{\mathcal{U}}(\Pi u: \kappa_1, \kappa_2) = \{ a \in \Lambda, \forall a_1 \in C^F_{\mathcal{U}}(\kappa_1), a_1 \in C^F_{\mathcal{U},t:a_1}(\kappa_2) \} \)
3. \( C^F_{\mathcal{U}}(\Pi k: u_1, \kappa_1) = \cap_{C \in CR(u_1)\kappa} C^F_{\mathcal{U},t:k:C}(\kappa_1) \)
4. \( C^F_{\mathcal{U}}(\Pi z: KSCM, \kappa_1) = \cap_{u_1 \in G \cap \text{arity}(u, \text{Kind})} C^F_{\mathcal{U},t:k:u_1}(\kappa_1) \)
5. \( C^F_{\mathcal{U}}(\kappa_2 \tau) = C^F_{\mathcal{U}}(\kappa_1) C^F_{\mathcal{U}}(\tau) \)
6. \( C^F_{\mathcal{U}}(\kappa_1, \kappa_2) = C^F_{\mathcal{U}}(\kappa_1) C^F_{\mathcal{U}}(\kappa_2) \)
7. \( C^F_{\mathcal{U}}(\lambda \tau: \kappa_1, \kappa_2) = a \in \Lambda \implies C^F_{\mathcal{U},t:a}(\kappa_2) \)
8. \( C^F_{\mathcal{U}}(\lambda \kappa: u_1, \kappa_1) = C \in CR(u_1)\kappa \implies C^F_{\mathcal{U}}(\kappa_1) \)
9. \( C^F_{\mathcal{U}}(\text{Ind}(k: \text{Kind})\{ \tilde{z} \}) = \) the least fixed point of the function \( F \) from \( \rho_0(\text{Kind}) \) to \( \rho_0(\text{Kind}) \) defined as:
   for all \( S \in \rho_0(\text{Kind}) \), for all \( C' \) in \( CR(I \rightarrow \text{Kind})_K \) (where \( I = \text{Ind}(k: \text{Kind})\{ \tilde{z} \} \)), for all sequences of pure terms \( b_i \), with for all \( i \),
   \( b_i \in C^F_{\mathcal{U},t:k:A', C': CR(\tilde{z})}(\llcorner \kappa_1, A', B') \)
   \( F(S) \) is the union of \( \text{min}(\text{Kind}) \) with the set of pure terms \( a \) such that
   \( \text{match} \ t, \{ C^F_{\mathcal{U},t:a_1}, b_{i}(\llcorner \kappa_1, a_1, \llcorner \lambda \tilde{z}, t a_2) \} \) \( a \in C' \) \( a \)
10. \( C^F_{\mathcal{U}}(\text{Elim}(\kappa, u)(\tau)\{ \tilde{z} \}) = G(C^F_{\mathcal{U}}(\kappa)) \)
where \( \kappa = \text{Ind}(k: \text{Kind})\{ \tilde{z} \} \) is well constructed of schema \( \text{Kind} \) under \( K \) and \( \mathcal{U} \) and \( G(C) \in \rho(u)\kappa \) is defined for all \( C \in \text{dom}(\prec_{\kappa}) \) as follows (\( \prec_{\kappa} \) is the order induced by the inductive definition \( \kappa \)):
   - If \( C^F_{\mathcal{U}}(\tau) \) has a normal form \( b = Co(\tilde{z}) \tilde{a} \) such that \( b \in C \)
     \( G(C) = C^F_{\mathcal{U},t_1, \mathcal{U}(\text{pred}(\tilde{b}))}(\llcorner \kappa_1, \kappa_1') \)
   - Can(\( u \)) otherwise

**Lemma 40** The function \( F \) in Definition 57.9 is monotonic.
Proof. We must prove that if $C_1 <_{\text{Kind}} C_2$, then
\[ C_{\mathcal{U}'}(C_{\mathcal{U}'}(A', B')) <_{\text{Kind}} C_{\mathcal{U}'}(C_{\mathcal{U}'}(A', B')) \]
The proof is by induction on the fact that $\kappa_i$ is the kind of a constructor.
- If $\kappa_i = \kappa$, then both sides reduce to $C' \circ \kappa(i)$.
- If $\kappa_i = \Pi X : A_1, A_2$, then it follows directly from the induction hypothesis and because $k$ does not occur in $A_1$.
- If $\kappa_i = \Pi X : A, k \to A_2$, then
\[ \zeta_{k,i}(\kappa_i, A', B') = \Pi Z : (\Pi X : A, k). \Pi X' : A. (A' (Z X')) \to \zeta_{k,i}(A_2, A', B' Z) \]

Suppose $\mathcal{U}' = \mathcal{U}, k : C', A' : C', B' : \kappa(i)$ where $C'$ is either $C_1$ or $C_2$. The required set is then
\[ a \in \Lambda, \text{ such that } \forall a_1 \in C_{\mathcal{U}'}(\Pi X : A, k), \forall a_2 \in C_{\mathcal{U}'}(\Pi X : A, A' (Z X')) \exists a_1 a_2 \in C_{\mathcal{U}'}(\Pi X : A, A', B' Z) \]
The set of $a_1$ and $a_2$ is larger for the LHS. By the induction hypothesis, the result $a a_1 a_2$ must occur in a smaller set for the LHS. The required result follows from this.

Remark 58. The previous lemma ensures that the interpretation of an inductive type sets up a well-defined order. This ensures that the interpretation of large elimination (Definition 57.10) is well-formed.

We get a bunch of substitution lemmas. The proof for each of these is similar and follows directly by induction over the structure of $\kappa$. We state them below:

Proposition 59. Let $\kappa$ be a well-constructed kind of schema $u$ under $\mathcal{K}$ and $\mathcal{U}$. Let $t$ be a type variable, and $\tau$ a type. We have that
\[ C_{\mathcal{U}}(\tau \sqcup t) = C_{\mathcal{U}}(\sigma) \]

Proposition 60. Let $\kappa$ be a well-constructed kind of schema $u$ under $\mathcal{K}$ and $\mathcal{U}$. Let $k$ be a kind variable and $\kappa_1$ a kind such that $\kappa_1$ is well-constructed under $\mathcal{K}$ and $\mathcal{U}$ of the same schema as $\mathcal{U}(k)$. We have that
\[ C_{\mathcal{U}}([k_1 / k]) = C_{\mathcal{U}}(\kappa_1) \]

Proposition 61. Let $\kappa$ be a well-constructed kind of schema $u$ under $\mathcal{K}$ and $\mathcal{U}$. Let $z$ be a schema variable, and $u_1$ be a schema such that $\mathcal{K}(u_1)$ is an arity. We have that
\[ C_{\mathcal{U}}([u_1 / z]) = C_{\mathcal{U}}(\kappa_1) \]

D.3.7 Candidate interpretation of kinds

Definition 62. We say that $\mathcal{U}$ and $\mathcal{U}'$ are equivalent interpretations if for all $\kappa$, we have that $\mathcal{U}(k) \simeq \mathcal{U}'(k)$ and for all $t$ we have that $\mathcal{U}(t) =_{\mathcal{U}} \mathcal{U}'(t)$.

Lemma 41. Let $u$ be a schema, $\mathcal{K}$ be a mapping, and $\mathcal{U}$ and $\mathcal{U}'$ be two equivalent interpretations. Suppose $\kappa$ is well-constructed of schema $\mathcal{K}(u)$ under $\mathcal{K}$ and both $\mathcal{U}$ and $\mathcal{U}'$. Then
\[ C_{\mathcal{U}}(\kappa) \simeq C_{\mathcal{U}}(\kappa) \]

Proof. The proof is by induction over the structure of $\kappa$. Most of the cases follow directly from the induction hypothesis.

- $\kappa = \text{Elim}(\kappa', \nu(u)) \{ \kappa' \}$. Here $\kappa' = \text{Ind}(k : \text{Kind}) \{ \kappa' \}$. First note that $C_{\mathcal{U}}(\kappa') = C_{\mathcal{U}}(\kappa')$. Therefore, the function $F$ whose $F_{U'}$ generates the inductive definition is the same. Moreover, $C_{\mathcal{U}}(\kappa) =_{\mathcal{U}} C_{\mathcal{U}}(\kappa)$. Since the set of pure terms is confluent, $C_{\mathcal{U}}(\kappa)$ and $C_{\mathcal{U}}(\kappa)$ have the same normal form. We can now do induction on the structure of $\kappa$ to prove that
\[ C_{\mathcal{U}}(\kappa) \simeq C_{\mathcal{U}}(\kappa) \]

Lemma 42. Let $\mathcal{K}$ be a mapping, $\mathcal{U}$ a candidate interpretation, $\kappa$ be a kind and $u$ be a schema such that $\kappa$ is a well-constructed kind of schema $\mathcal{K}(u)$. Then $C_{\mathcal{U}}(\kappa) \in \mathcal{C}(u)_\kappa$.

Proof. The proof is by induction over the structure of $\kappa$. Most of the cases follow in a direct way.

- $\kappa = \text{Ind}(k : \text{Kind}) \{ \kappa' \}$. We will use lemma 38 to prove this. For $S \in \mathcal{C}(u)_\kappa$, the first condition is satisfied by lemma 35.

Suppose $S = \bot$. If none of the branches is recursive then the function $F$ is a constant function and the proof is similar to the non-bottom case. Suppose the $\omega$th branch is recursive. Then it is easy to see that the $\omega$th branch is recursive. Then it is easy to see that the $\omega$th branch is recursive. Then it is easy to see that the $\omega$th branch is recursive.

- Consider any other $S$. We will show that $F(S)$ satisfies the conditions in Definition 33 and hence belongs to $\mathcal{C}(u)_\kappa$. $F(S)$ is defined as the union of $\mathcal{min}(\text{Kind})$ with the set of pure terms $a$ such that
\[ (\text{match } t. \{ C_{\mathcal{U}}(u_1) \{ a \} \{ \mathcal{Y}(\kappa_1, \lambda t, t_2) \} \} : C a) \]

This implies that $F(S)$ satisfies the conditions in Definition 33 and hence belongs to $\mathcal{C}(u)_\kappa$. $F(S)$ is defined as the union of $\mathcal{min}(\text{Kind})$ with the set of pure terms $a$ such that

Since $C$ is a candidate, the terms $a$ must be strongly normalizing.

To see that the set is closed under reduction, suppose $a \Rightarrow a'$. Since $C$ is a candidate we have that $(\text{match } t. \{ \ldots \} : C a) \Rightarrow C a'$. Moreover, we have that $C a = C a'$. Therefore, $a'$ is also in the generated set.

Suppose $a$ is a neutral term and for all $a'$ such that $a \Rightarrow a'$, we have that $a'$ belongs to this set. We have to prove that $a$ belongs to this set. This implies that we must prove:

Since $a$ is a neutral term, the above term does not have a redex at the head. From the induction hypothesis, we get that $C_{\mathcal{U}}(K,S,A',C,B') \circ \kappa(i)$ is a candidate and therefore closed under reduction. Moreover, the $b_i$ are strongly normalizing. We can now consider all possible reducts and prove by induction over $\nu(b_i)$ that the above condition is satisfied.
Definition 63 Suppose $\Delta$ is a context and $\mathcal{K}$ and $\mathcal{U}$ are a mapping and an interpretation. We say that $\mathcal{K}$ and $\mathcal{U}$ are adapted to $\Delta$ if:

- $\forall z \in \Delta$, we have that $\mathcal{K}(z)$ is an arity and $\vdash \mathcal{K}(z) : \text{Kind}$.
- $\forall k \in \Delta$, we have that $\mathcal{U}(k) \in \text{CR}(\Delta(k))$.
- $\forall t \in \Delta$, we have that $\mathcal{U}(t) \in C^\mathcal{U}_t(\Delta(t))$.

D.3.8 Interpretation of abstractions

We get a bunch of lemmas that state that an abstraction at the type level belongs to the corresponding kind. The proof of each of these lemmas is straightforward and follows in a similar way. We will show the proof for only one of the lemmas.

Lemma 43 Let $\Delta \vdash \lambda t : \kappa. \tau : \Pi t : \kappa. \kappa_1$ be a judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and a candidate interpretation adapted to $\Delta$. We have $C^\mathcal{K} u(\lambda t : \kappa. \tau : \Pi t : \kappa. \kappa_1)$ if and only if for all pure terms $a \in C^\mathcal{K} \kappa$. We have $C^\mathcal{K} u(\lambda t : \kappa. \tau : \Pi t : \kappa. \kappa_1)$ if and only if for all reducibility candidates $C \in \text{CR}(u)_\mathcal{K}$ we have that $C^\mathcal{K} u(\lambda t : \kappa. \tau : \Pi t : \kappa. \kappa_1)$.

Lemma 44 Let $\Delta \vdash \lambda k : u. \tau : \Pi k : u. \kappa$ be a judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and a candidate interpretation adapted to $\Delta$. We have $C^\mathcal{K} u(\lambda k : u. \tau : \Pi k : u. \kappa_1)$ if and only if for all reducibility candidates $C \in \text{CR}(u)_\mathcal{K}$ we have that $C^\mathcal{K} u(\lambda k : u. \tau : \Pi k : u. \kappa_1)$.

Lemma 45 Let $\Delta \vdash \lambda z : \text{Kscm}. \tau : \Pi z : \text{Kscm}. \kappa$ be a judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and a candidate interpretation adapted to $\Delta$. We have $C^\mathcal{K} u(\lambda z : \text{Kscm}. \tau : \Pi z : \text{Kscm}. \kappa_1)$ if and only if for all $u \in \text{aryt}(u)_\mathcal{K}$ we have that $C^\mathcal{K} u(\lambda z : \text{Kscm}. \tau : \Pi z : \text{Kscm}. \kappa_1)$.

Proof By definition $C^\mathcal{K} u(\lambda z : \text{Kscm}. \tau) = C^\mathcal{K} u(\tau)$. Similarly $C^\mathcal{K} u(\Pi z : \text{Kscm}. \kappa) = \cap_{u_1 \in \text{aryt}(u)_\mathcal{K}} C^\mathcal{K}_{u_1}(\kappa)$. If the part follows directly from the definition.

For the only if, suppose that $C^\mathcal{K} u(\tau) \in C^\mathcal{K}_{u_1}(\kappa)$ for all arities $u$. This implies that $C^\mathcal{K} u(\tau) \in \cap_{u_1 \in \text{aryt}(u)_\mathcal{K}} C^\mathcal{K}_{u_1}(\kappa)$. This implies that $C^\mathcal{K} u(\tau) \in C^\mathcal{K} u(\Pi z : \text{Kscm}. \kappa)$.

D.3.9 Interpretation of weak elimination

For this section $\kappa = \text{Ind}(\kappa : \text{Kind})[\kappa]$. Suppose also that $C \in \text{CR}(\kappa \rightarrow \text{Kind}) \mathcal{K}$ and $\tau_1 \in C^\mathcal{K}_{u_1,A',C'B'}(\mathcal{K},I(t_1, \kappa, A', B'))$.

Lemma 46 Suppose $a \in C^\mathcal{K} (\kappa)$. We have then

\[ \text{(match } t_1 \{ \tau_1, \tau_2, t_2 \}) \in C^\mathcal{K} \kappa \]

Proof Follows immediately from the definition of $C^\mathcal{K} (\kappa)$.

Lemma 47 Let $\Delta \vdash \text{Ind}(\kappa, \kappa_1) (\tau_1, \tau_2) : \kappa_2$ be a derivable judgment where $\kappa_1$ is a kind. Suppose $\mathcal{K}$ is a mapping and $\mathcal{U}$ is a candidate interpretation adapted to $\Delta$. If $C^\mathcal{K}_t(\tau_1) \in C^\mathcal{K}_t(\kappa_1, \kappa_1, \kappa_1, \text{Ctor } (i, k))$, then we have

\[ C^\mathcal{K}_t(\text{Ind}(\kappa, \kappa_1)(\tau_1, \tau_2)) \in C^\mathcal{K}_t(\kappa_1) \]

Proof Follows now from the previous lemma.

D.3.10 Interpretation of constructors

For this section, suppose $I = \kappa = \text{Ind}(\kappa : \text{Kind})[\kappa]$. Also, suppose $C \in \text{CR}(I \rightarrow \text{Kind})\mathcal{K}$.

Lemma 48 For all $i$, $\text{Co}(i) \in C^\mathcal{K}_t(\kappa_i)$.

Proof We know that $\kappa_i$ is of the form $\Pi X : A. k$. Suppose $\bar{B} \in C^\mathcal{K}_t, k C^\mathcal{K}_t(i) X : A$. Then we need to prove that $\text{Co}(i) \bar{B} \in C^\mathcal{K}_t(I)$. This means that we need to prove that

\[ \text{(match } t_1 \{ T(\kappa_i, a_i, \lambda \bar{t}_2) \}) \in C^\mathcal{K}_t(\bar{B}) \]

where $a_i$ belongs to the appropriate candidate. This implies that we need to prove that

\[ T(\kappa_i, a_i, \lambda \bar{t}_2, t_2) \in C^\mathcal{K}_t(\bar{B}) \]

This follows directly by an induction over the structure of $\kappa_i$.

D.3.11 Invariance under $\beta$ reduction

In this section, we show that the interpretation of kinds remains invariant under $\beta$ reduction.

Lemma 49 Let $\kappa$ be a well constructed kind of schema $u$ under a mapping $\mathcal{K}$ and candidate interpretation $\mathcal{U}$. If $\kappa \vdash \beta \kappa'$, then $\kappa'$ is well constructed of schema $u$ under $\mathcal{K}$ and $\mathcal{U}$, and $C^\mathcal{K}_t(\kappa') = C^\mathcal{K}_t(\kappa')$.

Proof The proof is by induction over the structure of $\kappa$. Most of the cases follow directly from the induction hypothesis. We will only consider $\beta$ reductions at the head.

- $\kappa = (\lambda t : \kappa_1, \kappa_2) \tau$. By definition,

\[ C^\mathcal{K}_t((\lambda t : \kappa_1, \kappa_2) \tau) = C^\mathcal{K}_t(\lambda t : \kappa_1, \kappa_2) C^\mathcal{K}_t(\tau) \]

Again by definition this is equal to $C^\mathcal{K}_t((\tau l) t' k_2)$.

- $\kappa = (\lambda k : u_1, \kappa_2) \beta \kappa_2$. By definition,

\[ C^\mathcal{K}_t((\lambda k : u_1, \kappa_2) \beta \kappa_2) = C^\mathcal{K}_t(\lambda k : u_1, \kappa_2) C^\mathcal{K}_t(\kappa_2) \]

By lemma 41 we have that $C^\mathcal{K}_t(\kappa_2) \in \text{CR}(u_1)_\mathcal{K}$. Therefore, we get that

\[ C^\mathcal{K}_t((\lambda k : u_1, \kappa_2) \beta \kappa_2) = C^\mathcal{K}_t(\lambda k : u_1, \kappa_2) C^\mathcal{K}_t(\kappa_2) \]

By proposition 60 this is equal to $C^\mathcal{K}_t([\kappa_2/\kappa_2]) \mathcal{K}$.
D.3.12 Invariance under \( \eta \) reduction

In this section, we show that the interpretation remains the same under \( \eta \) reduction. The unmarked terms \( \parallel \kappa \parallel \) are defined in Section D.2.1.

**Lemma 50** Let \( \kappa \) be a well-constructed kind of schema \( u \) under a mapping \( K \) and candidate interpretation \( U \). If \( \kappa \vdash_\eta \kappa' \), then \( \kappa' \) is well constructed of schema \( u \) under \( K \) and \( U \), and \( c^K_U(\kappa) = c^K_U(\kappa') \).

**Proof** The proof is again by induction over the structure of \( \kappa \).

- \( \kappa = \lambda t: \kappa_1. \left( \kappa_2 t \right) \). By definition \( c^K_U(\kappa) \) is equal to:

  \[
  a \in \Lambda \mapsto c^K_{U,t:a}(\kappa_2) \circ c^K_{U}(\kappa_1)(t)
  \]

  Since \( t \) does not occur free in \( \kappa_2 \), this is equivalent to

  \[
  a \in \Lambda \mapsto c^K_{U,t:a}(\kappa_2) \circ a
  \]

  Since \( a \) does not occur free now in \( c^K_U(\kappa_2) \), we get this is equivalent to \( c^K_U(\kappa_2) \). Note from Definition 34 that the domain of \( c^K_U(\kappa_2) \) is \( \lambda \).

- \( \kappa = \lambda k: u_1. \left( \kappa_2 k \right) \). By definition \( c^K_U(\kappa) \) is equal to:

  \[
  C \in CR(u_1)_K \mapsto c^K_{U,k:C}(\kappa_2) \circ c^K_{U,k:C}(k)
  \]

  Since \( k \) does not occur free in \( \kappa_2 \), this is equivalent to

  \[
  C \in CR(u_1)_K \mapsto c^K_{U,k:C}(\kappa_2) \circ C
  \]

  Since \( C \) does not occur free now in \( c^K_U(\kappa_2) \), we get this is equivalent to \( c^K_U(\kappa_2) \). Note from Definition 34 that the domain of \( c^K_U(\kappa_2) \) is \( CR(u_1)_K \).

\( \square \)

**Lemma 51** For all well-constructed kinds \( \kappa \) of schema \( u \) under \( K \) and \( U \), we have \( c^K_U(\kappa) = c^K_U(\parallel \kappa \parallel) \).

**Proof** Follows from the fact that \( \kappa = \beta_0 \parallel \kappa \parallel \).

\( \square \)

D.3.13 Invariance under \( \Delta \) reduction

In this section we essentially show that interpretation remains the same under large elimination.

**Lemma 52** Let \( \text{Elim}[\kappa, u](\tau) \left( \kappa' \right) \) be well constructed of schema \( K(u) \) under \( K \) and \( U \). Suppose \( \kappa = \text{Ind}(k: \text{Kind}(\kappa'))(\bar{r}) \). Suppose \( G \) is the function used for the interpretation of the large elimination. If \( c^K_U(\tau) \in c^K_U(\kappa) \), then for all \( C \in CR_0(\text{Kind}) \) with \( \text{Ind}(\kappa') \in C \), we have that \( G(c^K_U(\kappa)) = G(C) \).

**Proof** The proof is immediate.

\( \square \)

**Lemma 53** Suppose \( I = \kappa = \text{Ind}(k: \text{Kind}(\kappa'))(\bar{r}) \). Suppose the constructors of \( I \) are all small. Suppose the \( m \)th constructor of \( I \) has the form \( \Pi \bar{Y} : \bar{B}. k \) and we have a sequence of terms \( \bar{b} \) such that \( \text{Co}(m) \bar{b} \in c^K_U(I) \). Then we have that \( \bar{b}_i \in c^K_{U,k: \text{Ind}(k: \text{Kind}(\kappa'))}(\text{Co}(m) \bar{b})(B_i) \).

**Proof** We can have two cases.

- \( \text{pred}(\text{Co}(m) \bar{b}) \neq \perp \) This implies that \( \text{pred}(\text{Co}(m) \bar{b}) \in CR_0(\text{Kind}) \). Suppose \( S = c^K_{U,k: \text{Ind}(k: \text{Kind})(\text{Co}(m) \bar{b})}(B_i) \). Then we have that \( S \) is a candidate of schema \( \text{Kind} \). Suppose also that \( C' \) belongs to \( CR(I \rightarrow \text{Kind})_K \) and maps elements in the domain of \( I \rightarrow \text{Kind} \). Then for all indices \( i' \), we have that \( c^K_{U,(i \cdot \text{pred}(\text{Co}(m) \bar{b})) \cdot A_i}(\kappa, i', A', \text{Ctor}(i', I)) \) is a reducibility candidate of \( \text{Kind} \).

To prove the lemma we need to show that if for all indices \( i \)

\[
\tau_i \in c^K_{U,k: \text{Ind}(k: \text{Kind}(\kappa'))}(\text{Co}(m) \bar{b})), A_i, C_i'(\kappa, i', A', \text{Ctor}(i', I))
\]

then we have that \( \Phi_{k,1,1,B'}(\kappa, \tau, \tau_m) \) can reduce to \( \bar{b}_i \) by a head reduction. To have this, for the indices \( i \neq m \) choose \( \tau_i \) as some variable. For \( \tau_m \) choose the term that returns the \( i \)th argument of the constructor.

\[ \text{pred}(\text{Co}(m) \bar{b}) = \perp \text{ We can show that the constructors now are not recursive. Hence } k \text{ does not occur free in any of the } B_i. \] The proof for the previous case can be reused here.

\( \square \)

**Lemma 54** Let \( \Delta \vdash \text{Elim}[^\Delta [\kappa, u](\tau) \left( \kappa' \right) : u \) be a derivable judgment. Let \( K \) be a mapping and \( U \) be an interpretation adapted to \( \Delta \). Suppose \( I = \kappa = \text{Ind}(k: \text{Kind}(\kappa'))(\bar{r}) \). Suppose \( c^K_U(\tau) \in c^K_U(\kappa) \) and \( \tau \vdash_{\Delta} \text{Ctor}(i, \kappa, \bar{A}) \). Also suppose \( B = \lambda i' : I. \text{Elim}[\kappa, u](t) \left( \kappa' \right) \). We then have that \( c^K_U(\text{Elim}[\kappa, u](\tau) \left( \kappa' \right)) = c^K_U(\Phi_{k,1,1,B'}(\kappa, \bar{A})) \).

**Proof** Let \( G \) be the function used for interpreting large elimination. Suppose \( \text{Co}(i, \bar{a}) \) is the normal form of \( c^K_U(\tau) \). Then given the assumptions we have that:

\[
c^K_U(\text{Elim}[\kappa, u](\tau) \left( \kappa' \right)) = c^K_{U,B':G(\text{pred}(\text{Co}(i, \bar{a})))}(\Phi_{k,1,1,B'}(\kappa, \kappa')) (\bar{a})
\]

We therefore have to prove that

\[
c^K_{U,B':G(\text{pred}(\text{Co}(i, \bar{a})))}(\Phi_{k,1,1,B'}(\kappa, \kappa')) (\bar{a}) = c^K_U(\Phi_{k,1,1,B'}(\kappa, \kappa')) (\bar{A})
\]

- \( \kappa_1 = k \) it follows directly.

- \( \kappa_i = \Pi \righttriangleleft \kappa_1. \kappa_2 \) We have to prove that

\[
c^K_{U,B':G(\text{pred}(\text{Co}(i, \bar{a})))}(\Phi_{k,1,1,B'}(\kappa, \kappa')) (\bar{a}) = c^K_{U,t:a_1}(\Phi_{k,1,1,B'}(\kappa, \kappa')) (\bar{A}_{a_2}..n)
\]

Applying the induction hypothesis leads to the result.

- \( \kappa_i = \Pi \righttriangleleft \kappa \leftarrow \kappa_2 \) The LHS becomes

\[
c^K_{U,t:a_1}(\Phi_{k,1,1,B'}(\kappa, \kappa')) (\bar{A}_{a_2}..n)
\]

where \( U' = U, B' : G(\text{pred}(\text{Co}(i, \bar{a}))) \).

By lemma 53, \( a_1 \) belongs to \( c^K_U(\text{pred}(\text{Co}(i, \bar{a}))) \). This implies that \( \bar{a}_1 \bar{Y} \in \text{pred}(\text{Co}(i, \bar{a})) \). Moreover, by lemma 52 \( G(\text{pred}(\text{Co}(i, \bar{a}))) \) is equal to \( G(c^K_U(\kappa)) \) and which is in turn equal to \( c^K_U(\text{Elim}[\kappa, u](A_1, \bar{Y}))(\kappa')) \). The required result follows directly from here by performing one head reduction on the RHS and applying the induction hypothesis.

\( \square \)
D.3.14  Kinds invariant on their domain

**Definition 64** Let $\Delta \vdash \kappa : u$ be a derivable judgment and $K$ and $U$ be a mapping and an interpretation adapted to $\Delta$. We say $(\kappa, u, \Delta, K, U)$ is invariant if:

- $u = \text{Kind}$ and for all $\kappa'$ such that $\kappa \overset{*}{\supset} \kappa'$, we have that $C^U_\kappa(\kappa) = C^U_{\kappa'}(\kappa')$;
- $u = \Pi t : k_1, u_1$ then for all derivable judgments $\Delta \vdash \tau : k_1$ and $C^U_\kappa(\tau) \in C^U_{\kappa'}(k_1)$, we have that $(\kappa, \tau, [\tau/t]u_1, \Delta, K, U)$ is invariant;
- $u = \Pi k : u_1, u_2$ then for all derivable judgments $\Delta \vdash \kappa_1 : u_1$, we have that $(\kappa, k_1, [k_1/k]u_2, \Delta, K, U)$ is invariant.
- $u = z$ and we have that $(K(\kappa), K(u), \Delta(\kappa), \Delta, K, U)$ is invariant.

**Lemma 55** Let $\Delta \vdash \kappa_1 : \text{Kind}$ and $\Delta \vdash \kappa_2 : \text{Kind}$ be two derivable judgments and $K$ and $U$ be a mapping and an interpretation adapted to $\Delta$. If $(\kappa_1, \text{Kind}, \Delta, K, U)$ and $(\kappa_2, \text{Kind}, \Delta, K, U)$ are invariant and $\kappa_1 = \beta \kappa_2$, then $C^U_{\kappa_1}(\kappa_1) = C^U_{\kappa_2}(\kappa_2)$.

**Proof** We know that there exists a $B$ such that $\| \kappa_1 \| \overset{*}{\supset} B$ and $\| \kappa_2 \| \overset{*}{\supset} B$. This implies that there exists $\kappa'_1$ and $\kappa'_2$ (lemma 13 and 12) such that $\kappa_1 \overset{*}{\supset} \beta \kappa'_1$ and $\| \kappa'_1 \| \overset{*}{\supset} B$. Similarly, $\kappa_2 \overset{*}{\supset} \beta \kappa'_2$ and $\| \kappa'_2 \| \overset{*}{\supset} B$. From here we get that

$$C^U_{\kappa_1}(\kappa_1) = C^U_{\kappa_1}(\kappa'_1) = C^U_{\kappa_2}(\kappa'_2) = C^U_{\kappa_2}(\kappa_2).$$

The first reduction does not change the interpretation since we are reducing only a type. By lemma 54, the second does not change the interpretation. Finally, as above, we can prove that the result of the $\epsilon$ reduction is invariant over Kind.

D.3.16  Instantiations of contexts

**Definition 66** Let $\Delta$ be a well formed context. Let $\Theta$ be a context and $\phi$ be a mapping from variables to terms such that $\forall X \notin \Delta, \phi(X) = X$.

We say that $(\Theta, \phi)$ is an instantiation of $\Delta$ if for all variables $X \in \Delta$, we have that $\Delta \vdash \phi(X) : \phi(\Delta(X))$.

**Lemma 57** Let $\Delta \vdash A : B$ be a derivable judgment and $(\Theta, \phi)$ an instantiation of $\Delta$. Then $\Theta \vdash \phi(A) : \phi(B)$.

**Proof** By induction over the structure of $A$.

**Definition 67** (Adapted instantiation) We say that an instantiation $(\Theta, \phi)$ is adapted to a context $\Delta$ if:

- for all $t \in \Delta, \phi(t) \in C_{\text{Cano}(\Theta)}(\phi(\Delta(t)))$;
- for all $k \in \Delta, \phi(k) : \phi(\Delta(k)), \Theta, \emptyset, \text{Cano}(\Theta)$ is invariant;
- for all $z \in \Delta, \phi(z), Kscm, \Theta, \emptyset, \text{Cano}(\Theta)$ is invariant and $\phi(z)$ is an arity.

**Definition 68** Suppose $\Delta \vdash \kappa : u$ is a derivable judgment. We say that all instantiations of $(\kappa, u, \Delta)$ are invariant if for all instantiations $(\Theta, \phi)$ adapted to $\Delta$ and for all interpretations $U$ adapted to $\Theta$, we have that $(\phi(\kappa), \phi(u), \Theta, \emptyset, U)$ is invariant.

D.3.17  Kind schema invariant on their domain

**Definition 69** Let $\Delta \vdash u : Kscm$ be a derivable judgment and $K$ and $U$ be a mapping and an interpretation adapted to $\Delta$. We say that $(u, Kscm, \Delta, K, U)$ is invariant:

- if $u = \text{Kind}$, then $(u, Kscm, \Delta, K, U)$ is invariant;
- if $u = \Pi t : k_1, u_1$, then it is invariant if and only if $(k_1, \text{Kind}, \Delta, K, U)$ is invariant and for all terms $\tau$ such that $\Delta \vdash \tau : k_1$ is derivable and $C^U_\kappa(\tau) \in C^U_{\kappa_1}(k_1)$, we have that $(\tau/t)u_1, Kscm, \Delta, K, U)$ is invariant;
- if $u = \Pi \kappa : k_1, u_2$, then it is invariant if and only if $(k_1, \text{Kind}, \Delta, K, U)$ is invariant and for all terms $\tau$ such that $\Delta \vdash \tau : k_1$ is derivable and $C^U_{\kappa_1}(\tau) \in C^U_{\kappa_1}(k_1)$, we have that $(\tau/t)u_2, Kscm, \Delta, K, U)$ is invariant;
• if \( u = \Pi k : u_1, u_2 \), then it is invariant if and only if \((u_1, \text{Kscm}, \Delta, \mathcal{K}, \mathcal{U})\) is invariant, and for all kinds \( \kappa \) such that \( \Delta \vdash \kappa : u_1 \) is derivable and \( (\kappa, u_1, \Delta, \mathcal{K}, \mathcal{U}) \) is invariant, we have that \((\kappa, [k]/u_2, \text{Kscm}, \Delta, \mathcal{K}, \mathcal{U})\) is invariant;

• if \( u = z \), then it is invariant iff \((\mathcal{K}(z), \text{Kscm}, \Delta, \mathcal{K}, \mathcal{U})\) is invariant.

**Lemma 58** Let \( \Delta \vdash \kappa : u \) and \( \Delta \vdash u' : \text{Kscm} \) be derivable judgments. Let \( \mathcal{K} \) and \( \mathcal{U} \) be a mapping and an interpretation adapted to \( \Delta \). Suppose \( u = \beta_{\mathbb{U}} u' \), and \((u, \text{Kscm}, \Delta, \mathcal{K}, \mathcal{U})\) and \((u', \text{Kscm}, \Delta, \mathcal{K}, \mathcal{U})\) are invariant. If \((\kappa, u, \Delta, \mathcal{K}, \mathcal{U})\) is invariant, then \((\kappa, u', \Delta, \mathcal{K}, \mathcal{U})\) is also invariant.

**Proof** The proof is by induction over the structure of \( u \) and \( u' \).

• if \( u = u' = \text{Kind} \), then it is trivially true.

• if \( u = u' = z \), then again it is trivially true.

• if \( u = \Pi k : \kappa_1, u_1 \) and \( u' = \Pi k : \kappa_2, u_2 \), then we have that \( \kappa_1 = \beta_{\mathbb{U}} \kappa_2 \) and \( u_1 = \beta_{\mathbb{U}} u_2 \). By assumption, we know that \((\kappa_1, \text{Kind}, \Delta, \mathcal{K}, \mathcal{U})\) and \((\kappa_2, \text{Kind}, \Delta, \mathcal{K}, \mathcal{U})\) are invariant. This means that \( C^\kappa_{\mathcal{U}}(\kappa_1) = C^\kappa_{\mathcal{U}}(\kappa_2) \). Moreover, \( \Delta \vdash \tau : \kappa_1 \) is true iff \( \Delta \vdash \tau : \kappa_2 \) is true. Applying the induction hypothesis now leads to the required result.

• if \( u = \Pi k : u_1, u_2 \) and \( u' = \Pi k : u'_1, u'_2 \), the proof is similar to the previous case. \(\square\)

**Definition 70** Suppose \( \Delta \vdash u : \text{Kscm} \) is a derivable judgment. We say that all instantiations of \((u, \text{Kscm}, \Delta)\) are invariant if for all instantiations \((\Theta, \phi)\) adapted to \( \Delta \) and for all interpretations \( \mathcal{U} \) adapted to \( \Theta \), we have that \((\phi(u), \text{Kscm}, \Theta, \emptyset, \mathcal{U})\) is invariant.

**D.3.18 Strong normalization of pure terms**

**Theorem 71** Let \( \Delta \vdash \tau : \kappa \) be a derivable judgment and \( \mathcal{K} \) and \( \mathcal{U} \) be a mapping and an interpretation adapted to \( \Delta \). Then \( C^\kappa_{\mathcal{U}}(\tau) \in C^\kappa_{\mathcal{U}}(\kappa) \).

**Proof** The proof is by induction over the length of the derivation.

The induction hypothesis are as follows:

• if \( \Delta \vdash \tau : \kappa \) and \( \mathcal{K} \) and \( \mathcal{U} \) be a mapping and an interpretation adapted to \( \Delta \), then \( C^\kappa_{\mathcal{U}}(\tau) \in C^\kappa_{\mathcal{U}}(\kappa) \);

• if \( \Delta \vdash \kappa : u \), then all instantiations of \((\kappa, u, \Delta)\) are invariant;

• if \( \Delta \vdash u : \text{Kscm} \), then all instantiations of \((u, \text{Kscm}, \Delta)\) are invariant;

**type formation rules** This paragraph deals with rules of the form \( \Delta \vdash \tau : \kappa \).

• abstractions – Follows directly from the induction hypothesis and lemmas 43 and 44.

• var – Follows because the interpretation \( \mathcal{U} \) is adapted to the context \( \Delta \).

• weak elimination – Follows from lemma 47.

• constructor – Follows from lemma 48.

• weakening – Follows directly from the induction hypothesis since the mapping and interpretation remain adapted for a smaller context.

• conv – Follows from the recursion hypothesis and lemma 55.

• app – All three cases of app are proved similarly. We will show only one case here.

\[ \Delta \vdash \tau[u'] : \kappa. \] Then we know that \( \Delta \vdash \tau : \Pi z : \text{Kscm}, \kappa_1 \) and \( \Delta \vdash u' : \text{Kscm} \) and \([u'/z] \kappa_1 = \kappa \). By the induction hypothesis \( C^\kappa_{\mathcal{U}}(\tau) \in \mathcal{I} \). Suppose \( u'_1 = K(u') \). Then we know that \( C^\kappa_{\mathcal{U}}(\tau) \in C^\kappa_{\mathcal{U}}([u'/z] \kappa_1) \). But \( C^\kappa_{\mathcal{U}}([u'/z] \kappa_1) = C^\kappa_{\mathcal{U}}(\tau) \).

**kind formation rules** This paragraph deals with rules of the form \( \Delta \vdash \kappa : u \).

• product – All the product formation rules are proved in the same way. We show only one case here.

– Consider the following formation rule

\[ \Delta, z : \text{Kscm} \vdash \kappa : \text{Kind} \]

\[ \Delta \vdash \Pi z : \text{Kscm}, \kappa : \text{Kind} \]

We have to prove that for all instantiations \((\Theta, \phi)\) we have that \((\Pi z : \text{Kscm}, \phi(\kappa), \text{Kind}, \Theta, \emptyset, \mathcal{U})\) is invariant. This implies that we must prove that if \( \kappa \triangleright \kappa' \), then \( C^\phi_{\mathcal{U}}(\Pi z : \text{Kscm}, \phi(\kappa)) = C^\phi_{\mathcal{U}}(\Pi z : \text{Kscm}, \phi(\kappa')) \). By the induction hypothesis, for all instantiations \((\Theta, \phi, z : \text{arity}(u, \text{Kind}))\) we have that \((\phi, z : \text{arity}(u, \text{Kind}))(\kappa), \text{Kind}, \Theta, \emptyset, \mathcal{U})\) is invariant. This implies that if \( \kappa \triangleright \kappa' \) then \( C^\phi_{\mathcal{U}}(z : \text{arity}(u, \text{Kind}))(\kappa') = C^\phi_{\mathcal{U}}(z : \text{arity}(u, \text{Kind}))(\kappa') \).

The required result follows from here.

• var – follows since the instantiation is adapted.

• conv – follows from lemma 58.

• application – Both of the applications are proved similarly and follow directly from the induction hypothesis. We will show only one case here.

– If \( \Delta \triangleright \kappa_1 \kappa_2 : [\kappa_2/k]u \), then given \( \Theta, \phi, \text{and} \mathcal{U} \), we must prove that \((\phi(\kappa_1 \kappa_2), \phi([\kappa_2/k]u), \Theta, \emptyset, \mathcal{U})\) is invariant. But by the induction hypothesis we know that \((\phi(\kappa_1), \phi(\Pi k : u_1, u), \Theta, \emptyset, \mathcal{U})\) is invariant and \( \Delta \triangleright \kappa_2 : u_1 \). By lemma 57 \( \Theta \triangleright \phi(\kappa_2) = \phi(u_1) \). This leads to the required result.

• ind – Suppose \( I = \text{Ind}(k : \text{Kind})(\kappa) \). Note that \( C^\kappa_{\mathcal{U}}(I) \) depends only on \( C^\kappa_{\mathcal{U}}(k : S', A', B') \) where \( S \in \rho(\text{Kind}) \) and \( C \in \text{CR}(I \rightarrow \text{Kind}) \). By induction on the structure of \( \kappa \), we can show that this is invariant. This implies that if \( \kappa_i \triangleright \kappa'_i \), then the interpretation remains the same. If \( I \triangleright I' \), then for some \( i, \kappa_i \triangleright \kappa'_i \). From here we can deduce that if \( I \triangleright I' \), then \( C^\kappa_{\mathcal{U}}(I) = C^\kappa_{\mathcal{U}}(I') \).

• large elim – Follows from lemma 56.

• abstraction – Both of the abstractions are proved similarly. So we will show only one of the cases.
\[ \Delta \vdash \lambda \cdot \kappa_1, \kappa_2 : \Pi \vdash \kappa_1, u. \] We must prove that \( \phi(\lambda : \kappa_1, \kappa_2), \phi(\Pi : \kappa_1, u), \Theta, \emptyset, U \) is invariant, given \( \Theta, \phi, \) and \( U \). This implies that if \( \Theta \vdash \tau : \phi(\kappa_1) \) and \( \tau \) belongs to the appropriate candidate, then we must have \( \phi(\lambda : \kappa_1, \kappa_2), \tau, [\tau/t] \phi(u), \Theta, \emptyset, U \) is invariant. By proposition 65 we must prove that

\[
([\tau/t] \phi(\kappa_2), [\tau/t] \phi(u), \Theta, \emptyset, U)
\]
is invariant. But \( (\phi, t : \tau) \) is an instantiation that is adapted to \( (\Delta, t : \kappa_1) \). Applying the induction hypothesis now leads to the result.

**schema formation rules** This paragraph deals with rules of the form \( \Delta \vdash u : \text{Kscm} \).

- \( u = \text{Kind} \) follows directly.
- \( u = z \) follows since the instantiation is adapted.
- \( u = \Pi \vdash u_1, u_2 \) Given \( \Theta, \phi, \) and \( U \) we have to prove that \( \phi(\Pi : u_1, u_2), \text{Kscm}, \Theta, \emptyset, U \) is invariant. By the induction hypothesis, we know that \( \phi(\kappa_1), \text{Kscm}, \Theta, \emptyset, U \) is invariant. The induction hypothesis also says that \( \phi(\kappa_1), \text{Kscm}, \Theta, \emptyset, U \) is invariant. We also know that \( \Delta \vdash \kappa : \phi(u_1) \) and \( (\kappa, \phi(u_1), \Theta, \emptyset, U) \) is invariant since the instantiation is adapted. This implies that \( \phi(\kappa, \kappa_1), \text{Kscm}, \Theta, \emptyset, U \) is invariant.
- \( u = \Pi \vdash \kappa_1, u_1 \) the proof is very similar to the above case.

\[ \square \]

**Corollary 72** If \( \tau \) is a well formed type, \( |\tau| \) is strongly normalizing.

**Proof** Since \( \tau \) is well formed we have that \( \Delta \vdash \tau : \kappa \). We only need to construct an interpretation and a mapping. For the interpretation, let \( \mathcal{U}(t) = t \) for every type variable. Then we get \( c^{\mathcal{U}}_{\kappa}(\tau) = |\tau| \).

We can build the rest of \( \mathcal{U} \) and \( \mathcal{K} \) as:

- if \( \Delta = \cdot \) then \( \mathcal{U}(k) = \text{Can}_0(\text{Kind}) \) and \( \mathcal{K}(z) = \text{Kind} \) for all variables \( k \) and \( z \);
- if \( \Delta = \Delta', t : \kappa \) then return the \( \mathcal{U}' \) and \( \mathcal{K}' \) associated with \( \Delta' \);
- if \( \Delta = \Delta', k : u \) then \( \mathcal{U}' = \mathcal{U}, k : C \) and \( \mathcal{K}' = \mathcal{K} \), where \( C \in CR(u) \) and \( \mathcal{K}' \) and \( \mathcal{U}' \) are associated with \( \Delta' \);
- if \( \Delta = \Delta', z : \text{Kscm then } \mathcal{K}' = \mathcal{K}, z : \text{Kind and } \mathcal{U}' = \mathcal{U} \) where \( \mathcal{K}' \) and \( \mathcal{U}' \) are associated with \( \Delta' \).

\[ \square \]

**D.3.19 Normalization of terms**

In this section, we use an encoding that maps all well formed terms to types. This encoding preserves the number of reductions. The idea is similar to that of Harper et al [20].

The encoding uses two constants. \( A \) is a kind and \( B \) is a type.
- \( * \) is a variable that is never used, it is a wild-card.
- \( \text{A} : \text{Kind} \)
- \( \text{B} : \Pi k : \text{Kind}, k \)
- \( \text{*} \) unused variable

The encoding for \( \text{Kscm} \) is as follows:

\[ S(\text{Kscm}) = \text{Kscm} \]
\[ U(\text{Kscm}) = \text{Kind} \]
\[ K(\text{Kscm}) = A \]

The encoding for kinds is as follows:

\[ U(\text{Kind}) = \text{Kind} \]
\[ U(\Pi \vdash \kappa, u) = \Pi \vdash K(\kappa), U(u) \]
\[ U(\Pi \vdash u_1, u_2) = \Pi \vdash U(u_1), \Pi \vdash K(\kappa_1), U(u_2) \]
\[ U(z) = z \]
\[ K(\text{Kind}) = A \]
\[ K(\Pi \vdash \kappa, u) = \Pi \vdash K(\kappa), K(u) \]
\[ K(\Pi \vdash u_1, u_2) = \Pi \vdash U(u_1), \Pi \vdash K(\kappa_1), U(u_2) \]
\[ K(z) = k_2 \]

The encoding for terms is as follows:

\[ T(\Pi \vdash \kappa, u) = B[A \rightarrow \Pi \vdash \kappa, A \rightarrow A] \]
\[ T(\kappa) = \lambda t : A \rightarrow A \]
\[ T(z) = k_z \]

The encoding for kinds is as follows:

\[ K(k) = k \]
\[ K(\Pi \vdash \kappa_1, \kappa_2) = \Pi \vdash K(\kappa_1), K(\kappa_2) \]
\[ K(\Pi \vdash u, \kappa) = \Pi \vdash U(u), \Pi \vdash K(\kappa) \]
\[ K(\Pi \vdash \kappa, \kappa_1, \kappa_2) = \Pi \vdash K(\kappa_1), \Pi \vdash K(\kappa_2) \]
\[ K(\kappa, \tau) = K(\kappa, \tau) \]
\[ K(\kappa_1, \kappa_2) = K(\kappa_1), K(\kappa_2) \]

The encoding for terms is as follows:

\[ T(\text{Ind}(k : \text{Kind}) \{ \tau \}) = \text{Ind}(k : \text{Kind}) \{ K(\kappa) \} \]
\[ T(\text{Elim}[\kappa, u](\tau) \{ \kappa_1 \}) = \text{Elim}[K(\kappa), U(u)](T(\tau)) \{ K(\kappa) \} \]

The encoding for kinds is as follows:

\[ T(k) = k \]
\[ T(\Pi \vdash \kappa_1, \kappa_2) = B[A \rightarrow \Pi \vdash \kappa_1, A ightarrow A] \]
\[ T(\kappa) = \lambda t : A ightarrow A \]
\[ T(z) = z \]

The encoding for terms is as follows:

\[ T(\text{Ind}(k : \text{Kind}) \{ \tau \}) = \text{Ind}(k : \text{Kind}) \{ K(\kappa) \} \]
\[ T(\text{Elim}[\kappa, u](\tau) \{ \kappa_1 \}) = \text{Elim}[K(\kappa), U(u)](T(\tau)) \{ K(\kappa) \} \]
Lemma 59 For all well typed terms $A$, if $A \triangleright_{\beta} A'$, then we have

\[
T(A) \triangleright_{\beta} \, T(A')
\]

$K(A) \triangleright_{\beta} K(A')$

$U(A) \triangleright_{\beta} U(A')$

Moreover, if $\|A\| \triangleright_{\beta} A_1$, then there exists $A_2$ such that $\|A_2\| = A_1$ and $|T(A)| \triangleright_{\beta} |T(A_2)|$.

Lemma 60 For all well typed terms $A$, if $A \triangleright_{\eta} A'$, then we have

\[
T(A) \triangleright_{\eta} \, T(A')
\]

$K(A) \triangleright_{\eta} K(A')$

$U(A) \triangleright_{\eta} U(A')$

Moreover, if $\|A\| \triangleright_{\eta} A_1$, then there exists $A_2$ such that $\|A_2\| = A_1$ and $|T(A)| \triangleright_{\eta} |T(A_2)|$.

D.3.21 Coding and typing

In this section we show that the coding of a well typed term is also well typed. For this we need to prove that the coding preserves $\beta\eta$ equality. This requires a confluent calculus. Therefore, we use the unmarked terms from Section D.2.1. We extend the coding to unmarked terms by defining:

\[
\begin{align*}
U(\cdot) &= 1 \\
K(\cdot) &= 1 \\
T(\cdot) &= 1
\end{align*}
\]

Lemma 62 Suppose $\Delta \vdash A : B$ and $B \neq \text{Ext}$. Then we have that

\[
\Gamma(\Delta) \vdash T(A) : K(B) \text{ and } \Gamma(\Delta) \vdash T(A) : \text{Kind}
\]

Corollary 73 Suppose $\Delta \vdash A : B$ and $B \neq \text{Ext}$. Then $|T(A)|$ is strongly normalizing.

D.3.22 Normalization of unmarked terms

Lemma 63 For all well typed terms $A$, we have that $\|A\|$ is strongly normalizing for $\beta\eta\iota$ reduction.

Proof Since there can not be an infinite sequence of $\eta$ reductions and we can delay $\eta$ reductions, we need to prove the normalization for $\beta\eta\iota$ reductions only. Suppose $\|A\|$ is not normalizing and there exists a sequence $A_1 \ldots A_n \ldots$ such that $A_i \triangleright_{\beta\eta\iota} A_{i+1}$ and $A_0 = \|A\|$. By Lemma 59 and 60, we get that there exists a sequence of terms $A_1' \ldots A_n'$ such that $\|A_i'\| = A_i$ and $|T(A_i')| \triangleright_{\beta\eta\iota} |T(A_{i+1}')|$ and also $|T(A_i)| \triangleright_{\beta\eta\iota} |T(A_{i+1})|$. This implies that $|T(A)|$ is not strongly normalizing which is a contradiction. □

D.3.23 Normalization of all terms

Lemma 64 Suppose $A \triangleright_{\beta\iota} B$. Then $|T(A)| \triangleright_{\beta\eta\iota} |T(B)|$.

Proof By induction over the derivation of $A \triangleright_{\beta\iota} B$. Note that in taking a term $A$ to $T(A)$, all the terms $C$ that appear as annotations at lambda abstractions are duplicated with the corresponding $T(C)$. □

Lemma 65 Suppose $\Delta \vdash A : B$. Then $A$ is strongly normalizing.

Proof We only have to prove normalization for $\beta\iota$ reduction. By Lemma 64, if $A$ is not normalizing, then $|T(A)|$ is also not normalizing. But by lemma 62 we have that $\Gamma(\Delta) \vdash T(A) : K(B)$ which implies (lemma 63) that $|T(A)|$ is strongly normalizing. □

Theorem 74 (Strong normalization) All well typed terms are strongly normalizing.

Proof Follows from lemma 65. □

D.4 Church-Rosser Property

The proof is structured as follows:

- We first prove that a well typed term $A$ in $\beta\iota$ normal form has the same $\eta$ reductions as $\|A\|$.
- From here we know that if $A$ and $A'$ are in normal form, then $\|A\|$ and $\|A'|$ are equal. We then show that the annotations in the $\lambda$-abstractions are equal.

D.4.1 Structure of normal forms

Lemma 66 All well typed $\beta\iota$ normal terms $N$ have the following form:

1. $\lambda X : N_1, N_2$
2. $\Pi X : N_1, N_2$
Proof The proof is by induction over the size of $A$. We use lemma 66 to enumerate the different cases.

- The case where $A$ is a variable or a sort is immediate.
- Suppose $A, X : C, \Delta' \vdash A : B$. It follows directly from the induction hypothesis that $X$ does not occur in $N_1$ and $N_2$.
- Suppose $A, X : C, \Delta' \vdash \lambda Y : N_1, N_2 : B$ and $B = \Pi Y : N_1, N_2 : A'$. We know that $\Delta, X : C, \Delta' \vdash N_1 : s$ and therefore $X \notin FV(N_1)$. Also $B = \Pi Y : N_1, N_2 : A'$ and $\Delta, X : C, \Delta' \vdash Y : N_1 \cup N_2 : A''$. Since $N_1 \cup N_2 \notin FV(A'') \cup FV(N_1)$, we can apply the induction hypothesis and therefore $X \notin FV(N_2)$.
- Suppose $\Delta, X : C, \Delta' \vdash Y : N_1 : B$. This implies that $\Delta, X : C, \Delta' \vdash N_1 \cup N_2 : A'$ and the context $\Delta, X : C, \Delta' \vdash N_1 : A_1$. From lemma 23 and 13 we have that $\Delta, X : C, \Delta' \vdash N_1 : A_1$. From here we can show that $\Delta, X : C, \Delta' \vdash N_1 : A_3$. We can then apply the induction hypothesis to show that $X \notin FV(N_1)$. Iterating in this way, we can show that $X \notin FV(N_1)$.
- Suppose $\Delta, X : C, \Delta' \vdash \text{Ind}(Y : \text{Kind})\{\vec{N}\} : B$. Follows directly from the induction hypothesis that $X \notin FV(N_1)$.
- Suppose $\Delta, X : C, \Delta' \vdash \text{Ctor}(i, I) \vec{N} : B$. Follows directly from the induction hypothesis that $X \notin FV(I)$. We can then show as above that $X \notin FV(N_1)$.
- Suppose $\Delta, X : C, \Delta' \vdash \text{Elim}[N, N_1]\{\vec{N}\} \vec{N} : B$. Since $\Delta, X : C, \Delta' \vdash N : \text{Kind}$, it follows from the induction hypothesis that $X \notin FV(N)$. Similarly, since $\Delta, X : C, \Delta' \vdash N : \text{Kind}$, it follows that $X \notin FV(N_1)$. Similarly we can prove directly from the induction hypothesis that $X \notin FV(N_2) \cup FV(\vec{N})$. Finally, as above we can prove that $X \notin FV(\vec{N})$.  

Corollary 75 Let $\Delta \vdash A : B$. If $A$ is in normal form, then $\parallel A \parallel$ is also in normal form.

Proof We must show that $\parallel A \parallel$ does not contain any $\eta$ reductions. The interesting case is when $A$ is of the form $\lambda X : N_1, N_2 X$. We want to show that if $X \notin FV(\parallel N_2 \parallel)$, then $X \notin FV(N_2)$. Since it is well typed we know that $\Delta \vdash \lambda X : N_1, N_2 X : \Pi X : N_1, C$. We have that $X \notin FV(\Pi X : N_1, C)$. From here we get that $\Delta, X : N_1 \cup N_2 : \Pi X : N_1, C$. This implies that if $X \notin FV(\parallel N_2 \parallel)$, then $X \notin FV(N_2)$.

D.4.2 Church-Rosser

Theorem 76 (Church-Rosser) Let $\Delta \vdash A : B$ and $\Delta \vdash A' : B$ be two derivable judgments. If $A =_{\beta\eta} A'$, and if $A$ and $A'$ are in normal form, then $A = A'$.

Proof We know that $\parallel A \parallel$ and $\parallel A' \parallel$ are in normal form. Since the unmarked terms are confluent we have that $\parallel A \parallel \equiv \parallel A' \parallel$. The proof is by induction over the structures of $A$ and $A'$.

- The case when $A = A' = s$ or $A = A' = \text{a variable}$ is immediate.
- Suppose $A = \lambda X : N_1, N_2 A'$, and $A = \lambda X : N_1', N_2' A'$. We know that $N_1 =_{\beta\eta} N_1$, $N_2 =_{\beta\eta} N_2$, $N_1' =_{\beta\eta} N_1'$, $N_2' =_{\beta\eta} N_2'$. This implies that $N_1, N_2$ which implies that both of them have the same sort. This implies that $N_1 = N_2$. We can now apply the induction hypothesis to $N_2$ and $N_2^*$ to get that $N_2 = N_2^*$.
- Suppose $A = \Pi X : N_1, N_2 A'$, and $A = \Pi X : N_1', N_2' A'$. Follows directly from the induction hypothesis.
- Suppose $A = X \vec{N} A'$, and $A' = X \vec{N}'$. We know that in the context $\Delta$, the variable $X$ has the type $\Pi Y : N_1 A_3$. Therefore each of the $N_1$ and $N_1'$ have the same type. Applying the induction hypothesis to each of them leads to the required result.
- Suppose $A = \text{Ind}(X : \text{Kind})\{\vec{N}\}$ and $A' = \text{Ind}(X : \text{Kind})\{\vec{N}'\}$. By the typing rules we know that $\Delta, X : \text{Kind} \vdash N_1 = N_1'$. The induction hypothesis directly leads to $N = N'$. We can then show as above that $N_1 = N_1'$.
- Suppose $A = \text{Ctor}(i, N) \vec{N}$ and $A' = \text{Ctor}(i, N') \vec{N}'$. We know that both $N$ and $N'$ have type $\text{Kind}$. The induction hypothesis directly leads to $N = N'$. From here we can show that the $N_1$ and $N_1'$ are equal. Finally as above, we can show that the $N_{01}$ and the $N'_{01}$ are equal.

Theorem 77 (Confluence) Let $\Delta \vdash A : B$ and $\Delta \vdash A' : B$ be two judgments. If $A =_{\beta\eta} A'$, then $A$ and $A'$ have a common reduct – there exists a term $C$ such that $A \rightarrow^{*} C$ and $A' \rightarrow^{*} C$.

Proof We know that both $A$ and $A'$ reduce to normal forms $A_1$ and $A'_1$. Due to subject reduction, both $A_1$ and $A'_1$ have the same type $B$. By the previous lemma $A_1 = A'_1$.

D.5 Consistency

Theorem 78 (Consistency of the logic) There exists no term $A$ for which $\vdash A : \text{Kind}. X$.

Proof Suppose there exists a term $A$ for which $\vdash A : \Pi X : \text{Kind}. X$. By theorem 74, there exists a normal form $B$ for $A$. By the subject reduction $\vdash B : \Pi X : \text{Kind}. X$. We can show now that this leads to a contradiction by case analysis of the possible normal forms for the types in the calculus.