Design of the Certifying Programming Language
— Vero

Xinyu Feng
Email:xinyu.feng@yale.edu

June 3, 2004
Contents

1 Introduction ............................................. 5
  1.1 The Certifying Programming Language ................. 5
  1.2 Related Work ....................................... 7
  1.3 Design of Vero — Overview ........................... 8
  1.4 Contributions of My Work ............................. 8

2 The Internal Language ................................. 11
  2.1 The Proof Language — PL ............................. 11
    2.1.1 The PL Abstract Syntax ......................... 11
    2.1.2 Formalization .................................. 12
    2.1.3 Some Built-in PL Terms ......................... 13
  2.2 The Computation Language — CL ...................... 14
    2.2.1 The Abstract Syntax ............................ 14
    2.2.2 Static Semantics ................................ 15
    2.2.3 Dynamic Semantics .............................. 24
  2.3 Expressive Power of CL ............................. 25
    2.3.1 Higher-Kinded Recursive Types ................. 25
    2.3.2 Guarded Recursive Type Constructors .......... 28

3 The Vero Surface Language ............................ 37
  3.1 Lexical Conventions .................................. 37
    3.1.1 Notation Conventions ............................ 37
    3.1.2 White Space .................................... 37
    3.1.3 Comments ....................................... 37
    3.1.4 Identifiers and Operators ...................... 37
    3.1.5 Literals ........................................ 38
  3.2 Syntax .............................................. 40
  3.3 Sample Vero Programs ................................ 43
    3.3.1 Proof Terms .................................... 43
    3.3.2 CL Programs .................................... 47

4 Implementation ....................................... 49
  4.1 Implementing a PTS-like type system .................. 49
  4.2 Suspension-based lambda encoding ..................... 51
4.3 Hash-consing and memoization 	52
4.4 Other issues to worked on 	52

5 Conclusion and Future Work 	55

A Type Safety of CL 	61
Chapter 1

Introduction

1.1 The Certifying Programming Language

The idea of Proof-carrying code (PCC) [NL96, App01] 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. Here we propose a more general concept of certified program, which is a piece of code together with a proof that the code satisfies a given specification. It is more general than PCC because the code can be at any level, rather than low-level machine-language code; and the specification can be complex invariants and more advanced properties than safety property.

To implement the idea of certified program, we need tools to help us generate proofs about properties of programs, which include transforming unsafe programs to safe ones with proofs of the safety, compiling a certified high-level program down to certified binary programs, optimizing a certified program to get more efficient, but still certified software. Certifying Programming Language (CPL) is intended to be such a tool, which serves as a meta-language to write transformation tools for certified programs. Such a language should be able to represent the formal systems used to verify programs written in specific term languages, to reasoning about the encoding of the formal systems, and to analyse and manipulate the program and the proof terms.

There are a large number of formal verification frameworks are used these days, which range from type systems, Floyd-Hoare logic [Flo67, Hoa69], separation logic [Rey02], to TLA- or assumption-guarantee-based reasoning (of concurrent programs) [Lam94, Jon83]; each has been applied to different languages and has been very successful in its own domain. It is clear that there is no single framework that would fit all purposes. To support them all, we need incorporate in CPL a very expressive meta-logical framework (MLF). Each term language, whether it is a typed assembly language or a high-level language with Hoare-logic assertions, is implemented in CPL by first encoding the underlying
domain logic in MLF. The meta theory of each formal system used to verify these term languages can be verified in MLF.

On the other hand, to support the development of program transformation tools of the term language, such as certifying compilers, optimizers, etc., the CPL should support general-purpose programming idioms, e.g., side-effects, non-termination, exception handler, IO. Therefore the term languages can be represented in CPL by datatype definition, and the transforming tools is written as regular CPL programs. CPL type definitions can be indexed by MLF objects to ensure that the implementation of the term language accurately follows the underlying formal logic. Type checking for a term program can be done by simply checking its encoding in CPL (using the MLF and CPL checker).

Programming with certified binaries. To make the design of CPL more concrete, we give a quick overview on the key ideas. CPL provides first-class support to both general purpose data values and MLF objects; together they form another first-class structure called certified binaries [SSTP02]. A certified binary $S$ is a triple $(a, A, v)$ that consists of (1) an MLF term $a$ of type $d$ (where $d$ is an MLF representation type) (2) an MLF proof $A$ of $P(a)$ (where $P$ is a predicate on elements of type $d$), and (3) a CPL value $v$ of type $D$ (where $D$ is a CPL data type). The connection between $a$ and $v$ can be established by defining $D$ as a type indexed over terms of type $d$ [XP99, XCC03].

For example, we use $\text{Nat}$ to denote the natural number represented in MLF, and $\text{Prime}$ for the predicate that asserts an element of $\text{Nat}$ as a prime number. We introduce a singleton type constructor $s_{\text{Nat}}$: given an MLF term $n$ of type $\text{Nat}$, if a computation value $v$ has type $s_{\text{Nat}}[n]$, then $v$ denotes the natural number represented by $n$. A certified binary for a prime number contains three parts: an MLF term $n$ of type $\text{Nat}$, a proof for the proposition $\text{Prime}(n)$, and a CPL computation value of type $s_{\text{Nat}}[n]$. We can pack it up into an existential package and make it a first-class value with type:

$$\exists n : \text{Nat}. \exists t : \text{Prime}(n). s_{\text{Nat}}[n].$$

Notice that MLF objects are never directly dependent on CPL computation terms; in fact, because of this strong separation, a certified prime number package can still be implemented as a single integer at runtime.

We can also build certified binaries for programs that involve effects. Suppose $F$ is a CPL computation function that takes an integer as an argument and returns another as the result, and assume that $F$ may contain side-effects and may not always terminate. We can still reason about $F$ by first building a model $f$ inside MLF and then insist that $F$ has type:

$$\forall n : \text{Nat}. \ s_{\text{Nat}}[n] \to s_{\text{Nat}}[f(n)]$$

Here $f$ is an MLF function (which is always total) of type $\text{Nat} \to \text{Nat}$. The fact that $F$ has the above type does not mean that $F$ cannot have effects, rather, it only requires that the return value of $F$ (if it does terminate) is always equal to what is specified by the model $f$. 
1.2. RELATED WORK

We can show that \( F \) is a “certifying” implementation that always maps a prime number into another prime number by building an MLF proof object (for its model \( f \)) of type \( \forall t : \text{Nat}. \text{Prime}(t) \Rightarrow \text{Prime}(f(t)) \) where \( \Rightarrow \) denotes logical implication.

Of course, finding a model is often difficult, but in many cases we do not need to prove the partial correctness of \( F \). For example, assume \text{int} denotes the standard integer type. A function \( F \) of type:

\[
\text{int} \rightarrow \exists n : \text{Nat}. \exists t : \text{Prime}(n) . s_{\text{Nat}}[n].
\]

would already guarantee that the return value (if it returns) is always a prime number.

The above definitions can be easily generalized to more complex representation types. For example, if \( d \) is an MLF term representing a target programming language (TPL) and \( D \) is a CPL type definition (indexed over terms of type \( d \)), and suppose \( P \) is a security predicate showing that a term of \( d \) satisfies a particular safety policy, a certified TPL binary would simply consist of an MLF term \( a \) of type \( d \), an MLF proof of type \( P(a) \) showing that \( a \) is secure, and a CPL term (representing the actual TPL program) of type \( D(a) \). A “certifying” security rewriting function (implemented in CPL) is one that must have return type \( \exists a : d. \exists t : P(a) . D(d) \), that is, its implementation must also construct the proof certificate \( P(a) \) while inserting all the security checks.

1.2 Related Work

Although quite similar to the idea of PCC, the concept of certified program is more general, as explained in Section 1.1. There are other differences between the PCC framework and the Certifying Programming Language. First, PCC relies on a rather ad hoc process that uses external tools (e.g., meta-logical frameworks, verification condition generators) to construct proof certificates; a certifying programming language treats certified code as first-class citizens and provides direct support for creating and manipulating proof certificates. Second, existing PCC systems use a certifying compiler [CLN’00] to generate the object code and the safety certificate; the certificate can be checked by an external PCC checker, but there is no guarantee that the certifying compiler always generate a valid certificate. CPL fills this missing link by having both the code and the certificate be programmed in the same language—the validity of the certificate can be guaranteed by the CPL type system.

The research on CPL is based on the recent work on certified binaries (CB) by Shao, Saha, Trifonov and Papaspyrou [SSTP02], and the LTT system by Cray and Vanderwaart [CV02]. CB and LTT are mainly designed for compiler intermediate languages but the key ideas can be applied to the design of certifying programming languages. Both CB and LTT use singleton (or indexed) types [XP99] to support dependent types while maintaining a phase distinction between compile-time type checking and runtime evaluation. CB incorporates
the calculus of inductive constructions (used by Coq) [Pan98] while LTT uses the LF logical framework [HHP93].

1.3 Design of Vero — Overview

The language Vero is our first attempt to the design of a CPL. The name “vero” is a Latin word, which means “in truth, indeed, to be sure”. It reflects the objective of designing such a language.

The design of Vero is based on previous work of CB and LTT. Vero integrates the Calculus of Inductive Constructions (CIC) as the proof language (PL), with a general-purpose computation language (CL). Unlike CB, we have a separate type system for the computation language, while in CB it is inductively as part of the proof language. Such a design is quite similar to that of LTT. However, we use CIC instead of LF as the proof language, which is different with LTT.

Both the PL and the CL of Vero are explicitly typed language, that is because at present we put more emphasis on the expressiveness of our language, rather than user friendliness. Since we do not have to worry about decidable type inference, we have a very powerful type system for the computation language CL, which is much more general than that of SML. In addition to the core formulation presented in CB and LTT, we add the general recursive type constructor and the sum type to the type system of CL. Using the technique proposed by Collins and Shao [CS92], we have a natural way to define higher kinded recursive types. Also, with the introduction of the general recursive constructor and the sum type, we can easily simulate the guarded recursive datatype constructors proposed by Xi, Chen and Chen [XCC03].

Another difference between Vero and the previous work of CB and LTT is that Vero, as a CPL, is a surface language to be used directly by programmers, while CB and LTT are compiler intermediate languages. Therefore, in addition to the design of the internal language, we need to design the concrete syntax of the surface language of Vero.

The current version of Vero is just a prototype implementation. To focus on the implementation of such a powerful type system, we omitted a lot of orthogonal (but very important) issues, e.g. side-effects, exception handling, the module system, general pattern matching, and IO. All these issues will be researched in our future work.

1.4 Contributions of My Work

This report presents the design and implementation of the language Vero, the work of my 690 project. The project was carried out from January 2004 to April 2004 (in the fall of 2003 I was working on a different project, i.e. type checking BIOS code). The main contributions of my work are as follows:

1. I designed a richly typed internal language of Vero and formulated its static and dynamic semantics. The work is presented in Section 2.1 and
2.2 of Chapter 2. The type safety proof for CL is given in Appendix A.

2. I showed that by inductively define the Leibniz equality in our PL, we can simulate the guarded recursive datatype constructors proposed in [XCC03]. Such an ability allows us to have a uniform way to define the singleton datatypes indexed over the proof terms. I showed that the built-in singleton type $s_{\text{Bool}}$ in CB can be defined, and the complicated form of the if expression can be simulated using the more general case expression (see Section 2.3). Also I discussed some possible approaches to automatically generate the indexed type definition, given an inductive PL type.

3. I designed the concrete syntax of the Vero core language and presented some sample Vero programs, which is given in Chapter 3.

4. I implemented a type checker for Vero. Although the implementation is based on previous work of FLINT group [SLM98, Sah02], the fact that Vero is a surface language instead of a compiler intermediate language added extra complexity to the implementation; one example is we need provide helpful error messages for programmers to position the error. I also tried to improve the efficiency by memoizing more information than the previous implementation [SLM98, Sah02], although we have not test it over any benchmarks. The implementation is explained in detail in Chapter 4.
Chapter 2

The Internal Language

In this chapter we present the design of the internal language of Vero, which consists of a Proof Language (PL) used as a meta-logical framework, and a general purpose Computation Language (CL). In Section 2.3 we also show two important features of CL, i.e. the support of higher-kindred recursive types and guarded recursive type constructors.

2.1 The Proof Language — PL

We use the same type language proposed in [SSTP02], which is a minor revision of the Calculus of Inductive Constructions used in Coq [Tea03].

2.1.1 The PL Abstract Syntax

The proof language is in the form of PTS plus inductive definitions. Figure 2.1 shows the syntax of PL.

\[
\text{sort } s ::= \text{Prop} | \text{Type} | \text{Ext} \\
\text{ptm } A, B ::= s | X | X \cdot B | A \cdot B | \Pi X : A.B | \text{Ind}(X : B)\{\vec{A}\} | \text{Ctor}(i, A) \\
| \text{Elim}[A', B'](A)\{\vec{B}\}
\]

Figure 2.1: Syntax of the Proof Language

In addition to the standard features of PTS, PL also provides a general mechanism of inductive definitions [Pau93]. The term \text{Ind}(X : B)\{\vec{A}\} introduces an inductive type \(X\) with constructors whose types are listed in \(\vec{A}\). Here \(X\) must only occur “positively” inside each \(A_i\) (see [Pau93] for the formal definition of positivity). The term \text{Ctor}(i, A) refers to the \(i\)-th constructor in an inductive
CHAPTER 2. THE INTERNAL LANGUAGE

type $A$. For presentation, we will use a more friendly syntax in the rest of this paper. An inductive type $I = \text{Ind}(X : B)\{\bar{A}\}$ will be written as:

\[
\text{Inductive } I : B := c_1 : [I/X]A_1 \\
| c_2 : [I/X]A_2 \\
| .. \\
| c_n : [I/X]A_n
\]

We give an explicit name $c_i$ to each constructor, so $c_i$ is just an abbreviation of $\text{Ctor}(i, I)$.

PL provides iterators to support primitive recursion on inductive types. The eliminator $\text{Elim}[A', B'](A)(\bar{B})$ takes a term $A$ of inductive type $A'$, performs the iterative operation specified by $\bar{B}$ (which contains a branch for each constructor of $A'$), and returns a term of type $B'[A]$ (dependent elimination) or $B'$ (non dependent elimination) as the result.

Figure 2.2 gives a few examples of inductive definitions including the inductive types $\text{Bool}$ and $\text{Nat}$. The elimination for $\text{Nat}$ takes the following form $\text{Elim}[\text{Nat}, k](n)\{f_1 : f_2\}$. Here, $k$ is a dependent type with kind $\text{Nat} \rightarrow \text{Prop}$; $n$ is the argument which has type $\text{Nat}$. The term in the zero branch, $f_1$, has type $k$. The term in the succ branch, $f_2$, has type $\text{Nat} \rightarrow k n \rightarrow k n$. PL uses the $\iota$-reduction to perform the iterator operation. For example, the two $\iota$-reduction rules for $\text{Nat}$ work as follows:

\[
\text{Elim}[\text{Nat}, k](\text{zero})\{f_1 : f_2\} \rightarrow, f_1
\]
\[
\text{Elim}[\text{Nat}, k](\text{succ } n')\{f_1 : f_2\} \rightarrow, f_2 n' (\text{Elim}[\text{Nat}, k](n')\{f_1 : f_2\})
\]

Here we omitted the general $\iota$-reduction rule, which is formally defined in [SSTP02].

2.1.2 Formalization

We define the proof context $\Gamma$ as a list of bindings from variables to pseudo terms, which has the following syntax.

\[
\text{ctxt } \Gamma ::= \cdot | \Gamma, X : A
\]

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

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

The typing judgement for PL takes the form of $\Gamma \vdash A : B$. Since our PL is borrowed untouched from the TL of CB [SSTP02], we omit the typing rules for PL, which can be found in the Appendix of [SSTP02].

Proposition 2.1 and 2.2 are used in the type safety proof of CL in Section 2.2. They can be proved by simple induction on the PL typing rules.
2.1. **THE PROOF LANGUAGE — PL**

**Proposition 2.1** If $\Gamma_1, X : B, \Gamma_2 \vdash \text{Prop} : \text{Type}$ and $\Gamma_1 \vdash A : B$, then $\Gamma_1, [A/X] \Gamma_2 \vdash \text{Prop} : \text{Type}$.

**Proposition 2.2 (Proof Substitution)** If $\Gamma_1, X : B', \Gamma_2 \vdash A' : B'$, then $\Gamma_1, [A'/X] \Gamma_2 \vdash [A'/X] A : [A'/X] B$

### 2.1.3 Some Built-in PL Terms

In Figure 2.2, we present some PL terms, which are used in the type checking of the computation language in the following sections.

- **Inductive Bool** : Prop := true : Bool
  | false : Bool

- **Inductive Nat** : Prop := zero : Nat
  | succ : Nat → Nat

- **plus** : Nat → Nat → Nat

  - plus(zero) = $\lambda t : \text{Nat}. t$
  - plus(succ $t$) = $\lambda t' : \text{Nat}. \text{succ} ((\text{plus} \ t) \ t')$

- **ifez** : Nat → (Π $k : \text{Prop}$. k → (Nat → k) → k)

  - ifez(zero) = $\lambda k : \text{Prop}. \lambda t_1 : k. \lambda t_2 : \text{Nat} \rightarrow k. t_1$
  - ifez(succ $t$) = $\lambda k : \text{Prop}. \lambda t_1 : k. \lambda t_2 : \text{Nat} \rightarrow k. t_2 \ t$

- **le** : Nat → Nat → Bool

  - le(zero) = $\lambda t : \text{Nat}. \text{true}$
  - le(succ $t$) = $\lambda t' : \text{Nat}. \text{ifez} \ t' \ \text{Bool} \ (\text{le} \ t)$

- **lt** : Nat → Nat → Bool

  - $lt = \lambda t : \text{Nat}. \text{le} \ (\text{succ} \ t)$

- **Cond** : Bool → Prop → Prop → Prop

  - Cond(true) = $\lambda k_1 : \text{Prop}. \lambda k_2 : \text{Prop}. k_1$
  - Cond(false) = $\lambda k_1 : \text{Prop}. \lambda k_2 : \text{Prop}. k_2$

*Figure 2.2: Some PL terms*

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 addition to the inductive type Nat and Bool, plus is a function which calculates the sum of two natural numbers. The function ifez behaves like a switch statement: if its argument is zero, it returns a function that selects the first branch; otherwise, the result takes the second branch and applies it to the predecessor of the argument. The function le evaluates to
true if its first argument is less than or equal to the second. The function \( \text{lt} \) performs the less-than comparison. The definition of function \( \text{Cond} \) implements a conditional with result at the type level.

In the following sections, we use the meta-function \( \Downarrow \) to map natural numbers \( n \in \mathbb{N} \) to their representations as proof terms. It is defined inductively by \( \overline{0} = \text{zero} \) and \( n+1 = \text{succ} \overline{n} \), so \( \Gamma \vdash \overline{n} : \text{Nat} \) holds for all valid \( \Gamma \) and \( n \in \mathbb{N} \).

### 2.2 The Computation Language — CL

The computation language is an explicitly typed, general purpose programming language. In this report we just define the core language and we do not have a module system. Other language idioms like side-effects, exception handler and IO are orthogonal to our focus on an expressive type system. They are omitted to simplify the formulation.

#### 2.2.1 The Abstract Syntax

Figure 2.3 gives the syntax of the internal language of \( \text{vero} \).

\[
\begin{align*}
\text{kind} \quad \kappa & ::= \Omega \mid \kappa_1 \rightarrow \kappa_2 \mid \Pi X : A. \kappa \mid \kappa_1 \oplus \kappa_2 \\
\text{typ} \quad \tau & ::= \tau \mid \text{unit} \mid \text{unit}_\mathbb{N} \mid \text{unit}_\mathbb{B} \mid \tau_1 \rightarrow \tau_2 \mid \forall \tau : \kappa. \tau \mid \forall X : A. \tau \\
& \quad \mid \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \mid \tau_1 + \tau_2 \mid \mu(\tau_1, \tau_2) \mid \exists \tau : \kappa. \tau \mid \exists X : A. \tau \\
& \quad \mid \lambda \tau : X. \tau \mid \tau_1[\tau_2] \mid \Lambda X : A. \tau \mid \tau[A] \mid \langle \tau_1, \tau_2 \rangle \mid \pi_1 \tau \mid \pi_2 \tau \\
\text{fun} \quad f & ::= \lambda x : \tau. e \mid \Lambda t : \kappa. e \mid \Lambda X : A. e \\
\text{exp} \quad \pi & ::= x \mid () \mid \prod \mid e_1 \text{ aop } e_2 \mid \text{ tt } \mid \text{ ff } \mid e_1 \text{ cop } e_2 \\
& \quad \mid \text{ if } [B, A](e_0, X_1, e_1, X_2, e_2) \mid \text{ f } \mid \text{ fix } x : \tau. f \mid e_1 e_2 \mid e[\tau] \mid e[A] \\
& \quad \mid \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e[d] \mid \text{ inj}^0_\tau e \mid \text{ inj}^1_\tau e \mid \text{ case}(e_0, e_1, e_2) \\
& \quad \mid \text{ fold} \tau e \mid \text{ unfold } e \mid \langle t = \tau, e : \tau' \rangle \mid \langle X = A, e : \tau \rangle \mid \text{ let } x = e_1 \text{ in } e_2 \\
& \quad \mid \text{ let } t = \tau \text{ in } e \mid \text{ let } X = A \text{ in } e \mid \text{ let } \langle t, x \rangle = e_1 \text{ in } e_2 \\
& \quad \mid \text{ let } (X, x) = e_1 \text{ in } e_2 \\
\text{arith} \quad \text{ aop } ::= + \mid \ldots \\
\text{cmp} \quad \text{ cop } ::= < \mid \ldots \\
\text{val} \quad v ::= () \mid \prod \mid \text{ tt } \mid \text{ ff } \mid \lambda x : \tau. e \mid \Lambda t : \kappa. e \mid \Lambda X : A. e \mid \{ l_1 = v_1, \ldots, l_n = v_n \} \\
& \quad \mid \text{ inj}^0_\tau v \mid \text{ inj}^1_\tau v \mid \text{ fold} \tau v \mid \langle t = \tau, v : \tau' \rangle \mid \langle X = A, v : \tau \rangle \\
\end{align*}
\]

Figure 2.3: Syntax of the Computation Language
2.2. THE COMPUTATION LANGUAGE — CL

2.2.2 Static Semantics

The type context $\Delta$ is defined as a list of bindings from type variables to kinds, and the term context $\Sigma$ is a list of bindings from expression variables to types. They have the following syntax:

\[
\begin{align*}
tyctx & \quad \Delta ::= \cdot \mid \Delta, t : \kappa \\
tmctx & \quad \Sigma ::= \cdot \mid \Sigma, x : \tau
\end{align*}
\]

In Figure 2.4, we give a summary of the static semantics of the computation language.

\[
\begin{align*}
tyctx & \quad \text{formation} \quad \Gamma \vdash \Delta \ ok \\
tmctx & \quad \text{formation} \quad \Gamma; \Delta \vdash \Sigma \ ok \\
kind & \quad \text{formation} \quad \Gamma \vdash \kappa \ kind \\
type & \quad \text{formation} \quad \Gamma; \Delta \vdash \tau : \kappa \\
expr & \quad \text{formation} \quad \Gamma; \Delta; \Sigma \vdash e : \tau
\end{align*}
\]

Figure 2.4: Judgements

Contexts and Kind Formation

First we give the kind formation rules and type context formation rules.

\[
\begin{array}{ll}
\Gamma \vdash \kappa \ kind & \\
\hline
\Gamma \vdash \text{Prop} : \text{Type} & \quad \Gamma \vdash \Omega \ kind \\
\hline
\Gamma \vdash \kappa_1 \ kind \quad \Gamma \vdash \kappa_2 \ kind & \quad \Gamma \vdash \kappa_1 \rightarrow \kappa_2 \ kind \\
\hline
\Gamma \vdash \kappa_1 \ kind \quad \Gamma \vdash \kappa_2 \ kind & \quad \Gamma, X : A \vdash \kappa \ kind \\
\hline
\Gamma \vdash \Pi X : A.\kappa \ kind & \quad \Gamma \vdash \Delta \ ok \\
\hline
\Gamma \vdash \kappa \ kind & \quad \Gamma \vdash \kappa \ kind \\
\hline
\Gamma \vdash \Delta \ ok & \quad \Gamma \vdash \Delta, t : \kappa \ ok \quad (t \notin \text{dom}(\Delta)) \\
\hline
\Gamma \vdash \cdot \ ok & \quad \Gamma \vdash \Delta, t : \kappa \ ok \quad (\text{DEF-T})
\end{array}
\]

All types of CL expressions have kind $\Omega$. The kind $\kappa_1 \rightarrow \kappa_2$ is the kind of higher-order type constructors. Two mutually recursive defined type constructors has the kind $\kappa_1 \otimes \kappa_2$, which will be explained in detail in section 2.3. The kind $\Pi X : A.\kappa$ allows us define CL type constructors indexed over PL terms.
Proposition 2.3 and 2.4 are substitution propositions for the kind language.

**Proposition 2.3** If $\Gamma_1, X : B, \Gamma_2 \vdash \kappa$ kind and $\Gamma_1 \vdash A : B$, then $\Gamma_1, [A/X] \Gamma_2 \vdash [A/X]|\kappa$ kind.

**Proof.** By induction of the derivation of $\Gamma_1, X : B, \Gamma_2 \vdash \kappa$ kind (Proposition 2.1 is used). □

**Proposition 2.4**

1. If $\Gamma \vdash \Delta, t : \kappa$ ok, then $\Gamma \vdash \Delta$ ok.
2. If $\Gamma_1, X : B, \Gamma_2 \vdash \Delta$ ok and $\Gamma_1 \vdash A : B$, then $\Gamma_1, [A/X] \Gamma_2 \vdash [A/X] \Delta$ ok.

**Proof.** By induction on the derivations of $\Gamma \vdash \Delta_1, t : \kappa, \Delta_2$ ok and $\Gamma_1, X : B, \Gamma_2 \vdash \Delta$ ok respectively (Propositions 2.1 and 2.3 are used to prove 2). □

**Type Formation**

Figure 2.5 show the formations rules of types. Most of the types have standard semantics, except that $s_{\text{Nat}}$ and $s_{\text{Bool}}$ are singleton type constructors indexed by PL terms which have type $\text{Nat}$ and $\text{Bool}$, respectively. Rule T-ALL1 and T-ALL2 show that we have polymorphism with regard to both CL types and PL terms. The structure type shown in rule T-STR is basically record type: all the fields must have types in kind $\Omega$, and we do not have dependent.

The $\mu$ constructor, together with type tuples $(r_1, r_2)$ and projections, shown in rule T-REC, T-TUP, T-PRJL, and T-PRJR, are used to formulate recursive types, which are explained in detail in Section 2.3.

Existential types shown in rule T-EX1 and T-EX2 give us the power to introduce first-class modules in the system, which will be our future research. However, the existential type $\exists X : A. \tau$ also plays an important role to pack the proof terms together with computation terms, which is an essential feature in CL. In Section 2.3 we will show the usage of the existential types.

Type functions shown in rule T-FN1 and T-FN2 and corresponding application rules allow us define higher-order type constructors and constructors indexed over PL terms.
Figure 2.5: Type formation rules
Type Reduction and Equivalence

First we give the reducible types by the following rules.

\[(\lambda x : k. \tau)[\tau'] \sim_\iota [\tau'/x]_\tau\]  \hspace{2cm} (TR-\beta 1)

\[(\lambda x : B. \tau)[A] \sim_\iota [A/X]_\tau\]  \hspace{2cm} (TR-\beta 2)

\[\lambda x : k. \tau[x] \sim_\iota \tau \quad x \text{ is not free in } \tau\]  \hspace{2cm} (TR-\eta 1)

\[\lambda X : A. \tau[X] \sim_\iota \tau \quad X \text{ is not free in } \tau\]  \hspace{2cm} (TR-\eta 2)

\[\pi_1(\tau_1, \tau_2) \sim_\iota \tau_1\]  \hspace{2cm} (TR-\proj 1)

\[\pi_2(\tau_1, \tau_2) \sim_\iota \tau_2\]  \hspace{2cm} (TR-\proj 2)

In Figure 2.6 we define the type reduction relation \(\triangleright_\iota\). The \(\sim\) relation shown in rules PRRED-1 to PRRED-4 is the reduction relation between proof terms, which is the union of the \(\beta\)-reduction, \(\eta\)-reduction and \(\iota\)-reduction (denoted by \(\sim_\beta\), \(\sim_\eta\), and \(\sim_\iota\), respectively).

Definition 2.5 The multi-step reduction relation \(\triangleright^*_\iota\) and the number of reduction steps is inductively defined in the following way:

- \(\tau \triangleright^*_\iota \tau\) in 0 steps;
- if \(\tau_1 \triangleright^*_\iota \tau_2\) in \(k\) steps, and \(\tau_2 \triangleright_\iota \tau_3\), then \(\tau_1 \triangleright^*_\iota \tau_3\) in \(k + 1\) steps.

The type equivalence relation \(\equiv_\iota\) is defined by the rules shown in Figure 2.7.

The following propositions show properties of the type system of CL. Confluence (Proposition 2.7) and strong normalization (Proposition 2.11) are two critical properties for a sound type system. However, there are no short proofs for such properties. Intuitively, our type system has such properties because most of the constructs in the type system are standard and there are literatures that have proved these properties for similar systems. Although we have the \(\mu\) constructor to define recursive types, the folded form and unfolded form are not equivalent types. Also CL types only have dependency on PL terms, which is proved to be confluent and strong normalizing in [Sah02]. The proofs of Proposition 2.7 and 2.11 are omitted here.

Proposition 2.6

1. If \(\tau_1 =_\iota \tau_2\), for any \(\tau\) and \(t\) we have \([\tau/t]_\tau =_i [\tau/t]_\tau_1\).

2. If \(\tau_1 =_\iota \tau_2\), for any \(A\) and \(X\) we have \([A/X]_\tau_1 =_i [A/X]_\tau_2\).

Proof. Obvious... \(\Box\)

Proposition 2.7 (Confluence) If \(\tau \triangleright_\iota \tau_1\) and \(\tau \triangleright_\iota \tau_2\), there exists a \(\tau'\) such that \(\tau_1 \triangleright^*_\iota \tau'\) and \(\tau_2 \triangleright^*_\iota \tau'\).
Figure 2.6: Type reduction relation $\triangleright_{\downarrow}$
Proposition 2.8 (Substitution-1)  If $\Gamma; \Delta,t : \kappa' \vdash \tau : \kappa$, and $\Gamma; \Delta \vdash \tau' : \kappa'$, then $\Gamma; \Delta \vdash [\tau'/t] \tau : \kappa$.

Proof sketch. By induction on the derivation of $\Gamma; \Delta,t : \kappa' \vdash \tau : \kappa$ (Proposition 2.4 is used). ∎

Proposition 2.9 (Substitution-2)  If $\Gamma_1,X : B,\Gamma_2;\Delta \vdash \tau : \kappa$ and $\Gamma_1 \vdash A : B$ then $\Gamma_1,[A/X]\Gamma_2;[A/X]\Delta \vdash [A/X]\tau : [A/X]\kappa$.

Proof sketch. By induction on the derivation of $\Gamma_1,X : B,\Gamma_2;\Delta \vdash \tau : \kappa$ (Proposition 2.4 is used). ∎

Proposition 2.10 (Subject Reduction)  If $\Gamma; \Delta \vdash \tau : \kappa$ and $\tau \triangleright t \tau'$, we must have $\Gamma; \Delta \vdash \tau' : \kappa$.

Proof sketch. Here we give the sketch of the proof. First we need to prove the inversion of type formation rules. Then by induction on the $\leadsto_\tau$ rules, we prove that if $\Gamma; \Delta \vdash \tau : \kappa$ and $\tau \leadsto_\tau \tau'$, we must have $\Gamma; \Delta \vdash \tau' : \kappa$.

Finally by induction on the $\triangleright_\tau$ rules we prove the proposition. ∎

Proposition 2.11 (Strong Normalization)  For any type $\tau$, if $:\vdash \tau : \kappa$ for some $\kappa$, then there exists $n \geq 1$ such that there is no $\tau'$ such that $\tau \triangleright_\tau \tau'$ in $n$ steps.
2.2. THE COMPUTATION LANGUAGE — CL

Expression formation

First we give the formation rules of term context.

\[
\frac{}{\Gamma; \Delta \vdash \Sigma} \text{ ok}
\]

\[
\frac{\Gamma \vdash \Delta \quad \Gamma; \Delta \vdash \Sigma}{\Gamma; \Delta \vdash \Sigma \text{ ok}}
\]

\[
\frac{\Gamma; \Delta, x : \tau \vdash \Sigma \text{ ok}}{\Gamma; \Delta \vdash \Sigma, x : \tau \text{ ok}}
\] (INT-TMC)

\[
\frac{\Gamma; \Delta \vdash \Sigma \text{ ok} \quad \Gamma; \Delta \vdash \tau : \Omega}{\Gamma; \Delta \vdash \Sigma, x : \tau \text{ ok}}
\] (DEF-E) \( x \notin \text{dom(\Sigma)} \)

**Proposition 2.12**

1. If \( \Gamma; \Delta \vdash \Sigma, x : \tau \text{ ok} \), then \( \Gamma; \Delta \vdash \Sigma \text{ ok} \).

2. If \( \Gamma; \Delta, t : \kappa \vdash \Sigma \text{ ok} \) and \( \Gamma; \Delta \vdash \tau : \kappa \), then \( \Gamma; \Delta \vdash [\tau/t] \Sigma \text{ ok} \).

3. If \( \Gamma, X : B ; \Delta \vdash \Sigma \text{ ok} \) and \( \Gamma \vdash A : B \), then \( \Gamma ; [A/X] \Delta \vdash [A/X] \Sigma \text{ ok} \).

**Proof.** By induction on the derivation of \( \Gamma; \Delta \vdash \Sigma, x : \tau \text{ ok} \), \( \Gamma; \Delta, t : \kappa \vdash \Sigma \text{ ok} \), and \( \Gamma, X : B ; \Delta \vdash \Sigma \text{ ok} \), respectively (Propositions 2.8 and 2.9 are used to prove 2 and 3, respectively). \( \square \)

Figure 2.8 and 2.9 show the formations rules of expressions.

Rules E-NAT, E-TRUE, and E-FALSE assign singleton types to numeric and boolean constants. For instance the constant `True` has type `sNat([succ zero])` in any valid environment.

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-IT use respectively the type terms `plus` and `lt` (defined in Section 2.1.3) to reflect the semantics of the term operations into the type level via singleton types.

The branching construct `if[B,A](e,X_1,e_1,X_2,e_2)` allows information obtained dynamically (e.g., through comparisons) to be made available for static reasoning in the form of proof parameters to its branches. The type term `A` represents a proof of the proposition encoded by either `B true` or `B false`, depending on the value of `e`. This proof is bound to the type variable `(X_1 \text{ or } X_2)` of the appropriate branch, which can use it in the construction of other proofs. The correspondence between the value of `e` and the type of `A` is again established through a singleton boolean type. Thus for instance if the run-time value of `e` asserts the truthfulness of some proposition `P`, since the type parameter `A''` of the singleton type of `e` reflects the value of `e` at the type level, we can define `B` so that `B A''` represents `P` or `¬P`, depending on whether `A'' = B \text{ true}` or `A'' = B \text{ false}`, and reason in each of the two branches under the assumption that `P` or `¬P`, respectively. Of course, for this reasoning to be sound, we need a proof that `A''` indeed reflects the truthfulness of `P`, that is, we need a proof term `A` of kind `B A''`.

In fact if is more flexible than that, because `B false` does not have to be the negation of `B true`, one can have imprecise information flow into the branches.
CHAPTER 2. THE INTERNAL LANGUAGE

Γ; Δ; Σ ⊢ e : τ

Γ; Δ ⊢ Σ ok

Γ; Δ; Σ ⊢ x : τ

(E-VAR)

Γ; Δ; Σ ⊢ x : τ

(E-UNIT)

Γ; Δ; Σ ⊢ () : unit

(E-TRUE)

Γ; Δ ⊢ Σ ok

Γ; Δ; Σ ⊢ ⊤ : τ

(E-NAT)

Γ; Δ; Σ ⊢ sNat[A1]

Γ; Δ, Σ ⊢ e1, e2 : sNat[A2]

(E-ADD)

Γ; Δ; Σ ⊢ sNat[A1]

Γ; Δ, Σ ⊢ e1, e2 : sNat[A2]

(E-UNIT)

Γ; Δ; Σ ⊢ () : unit

Γ; Δ ⊢ Σ ok

Γ; Δ, Σ ⊢ tt : sBool[true]

(E-TRUE)

Γ; Δ ⊢ Σ ok

Γ; Δ; Σ ⊢ ff : sBool[false]

(E-FALSE)

Γ ⊢ B : Bool → Prop

Γ; Δ; Σ ⊢ e0 : sBool[A']

Γ ⊢ A : B A'

Γ, X1 : B true; Δ; Σ ⊢ e1 : τ

Γ ⊢ τ : Ω

Γ, X2 : B false; Δ; Σ ⊢ e2 : τ

(E-FP)

Γ; Δ, Σ ⊢ if[B, A](e0, X1, e1, X2, e2) : τ

Γ; Δ; Σ; x : τ1 ⊢ e : τ2

(E-FN)

Γ; Δ; Σ ⊢ λx : τ1.e : τ1 → τ2

(E-TFN1)

Γ ⊢ A : s

Γ; Δ; Σ ⊢ e1, e2 : τ1

Γ ⊢ τ : Ext

Γ; Δ, Σ ⊢ X : s

(E-TFN2)

Γ; Δ; Σ ⊢ ∀X : A.e : τ

Γ; Δ; Σ ⊢ λX : A.e : ∀X : A.τ

(E-FIX)

Γ; Δ; Σ ⊢ x : τ.f : τ

Γ; Δ; Σ ⊢ fix x : τ.f : τ

(E-APP)

Γ; Δ ⊢ e1 : τ1 → τ2

Γ; Δ; Σ ⊢ e2 : τ1

Γ; Δ; Σ ⊢ e1.e2 : τ2

(E-TAPP1)

Γ ⊢ A : B

Γ ⊢ B : s

Γ; Δ, Σ ⊢ e : ∀X : B.τ

Γ; Δ; Σ ⊢ e[A] : [A/X]τ

(E-TAPP2)

Γ; Δ; Σ ⊢ e1 : τi

Γ; Δ; Σ ⊢ e1 : τi

(E-STR)

Γ; Δ, Σ ⊢ i = 1, \ldots, n

Γ; Δ; Σ ⊢ e : \{l_1 : τ_1, \ldots, l_n : τ_n\}

Γ; Δ, Σ ⊢ e.i : τ_i

(E-PROJ)

Figure 2.8: Expression formation rules
\[ \Gamma; \Delta; \Sigma \vdash e : \tau \]

\[ \frac{\Gamma; \Delta \vdash t_1 + t_2 : \Omega \quad \Gamma; \Delta; \Sigma \vdash e : t_1}{\Gamma; \Delta; \Sigma \vdash \text{inj}^{t_1 + t_2}_1 e : t_1 + t_2} \quad (E\text{-inj}) \]

\[ \frac{\Gamma; \Delta \vdash t_1 + t_2 : \Omega \quad \Gamma; \Delta; \Sigma \vdash e : t_2}{\Gamma; \Delta; \Sigma \vdash \text{inj}^{t_1 + t_2}_2 e : t_1 + t_2} \quad (E\text{-inj}) \]

\[ \Gamma; \Delta; \Sigma \vdash e_0 : t_1 + t_2 \quad \Gamma; \Delta; \Sigma \vdash e_1 : t_1 \rightarrow \tau \quad \Gamma; \Delta; \Sigma \vdash e_2 : t_2 \rightarrow \tau \]

\[ \Gamma; \Delta; \Sigma \vdash \text{case}(e_0, e_1, e_2) : \tau \quad (E\text{-case}) \]

\[ \frac{\Gamma; \Delta \vdash \mu_\kappa(t_1, t_2) : \Omega \quad \Gamma; \Delta; \Sigma \vdash e : K(\kappa, t_1, t_2)}{\Gamma; \Delta; \Sigma \vdash \text{fold}_{\kappa}(t_1, t_2) e : \mu_\kappa(t_1, t_2)} \quad (E\text{-fold}) \]

\[ \frac{\Gamma; \Delta; \Sigma \vdash e : \mu_\kappa(t_1, t_2)}{\Gamma; \Delta; \Sigma \vdash \text{unfold} e : K(\kappa, t_1, t_2)} \quad (E\text{-unfld}) \]

\[ \frac{\Gamma; \Delta \vdash \tau : \kappa \quad \Gamma; \Delta; e : \kappa \vdash \tau' : \Omega \quad \Gamma; \Delta; \Sigma \vdash e : [\tau/\tau']}{\Gamma; \Delta; \Sigma \vdash \langle e = e' : \tau' \rangle : \exists \kappa. \tau'} \quad (E\text{-pack1}) \]

\[ \frac{\Gamma \vdash A : B \quad \Gamma \vdash X : B; \Delta \vdash \tau : \Omega}{\Gamma \vdash B : s \quad \Gamma \vdash \Delta; \Sigma \vdash e : [A/X]_{\tau}} \quad (s \neq \text{Ext}) \quad (E\text{-pack2}) \]

\[ \frac{\Gamma; \Delta; \Sigma \vdash e_0 : \tau_0 \quad \Gamma; \Delta, \Sigma, x : \tau_0 \vdash e : \tau}{\Gamma; \Delta; \Sigma \vdash \text{let} x = e_0 \text{ in } e : \tau} \quad (E\text{-let}) \]

\[ \frac{\Gamma; \Delta; \Sigma \vdash \tau_0 : \kappa \quad \Gamma; \Delta; \Sigma \vdash [\tau_0/t]e : \tau}{\Gamma; \Delta; \Sigma \vdash \text{let } t = \tau_0 \text{ in } e : \tau} \quad (E\text{-tlet1}) \]

\[ \frac{\Gamma \vdash A : B \quad \Gamma \vdash \Delta; \Sigma \vdash [A/X]e : \tau}{\Gamma \vdash \Delta; \Sigma \vdash \text{let } X = A \text{ in } e : \tau} \quad (E\text{-tlet2}) \]

\[ \frac{\Gamma; \Delta \vdash \tau : \Omega \quad \Gamma; \Delta; \Sigma \vdash e_0 : \exists \kappa. \tau_0 \quad \Gamma; \Delta, \Sigma, \tau : \tau_0 \vdash e : \tau}{\Gamma; \Delta; \Sigma \vdash \text{let}(t, x) = e_0 \text{ in } e : \tau} \quad (E\text{-open1}) \]

\[ \frac{\Gamma; \Delta \vdash \tau : \Omega \quad \Gamma; \Delta; \Sigma \vdash e_0 : \exists X. A \tau_0 \quad \Gamma; X : A; \Delta, \Sigma, x : \tau_0 \vdash e : \tau}{\Gamma; \Delta; \Sigma \vdash \text{let}(X, x) = e_0 \text{ in } e : \tau} \quad (E\text{-open2}) \]

\[ \frac{\Gamma; \Delta; \Sigma \vdash \tau : \tau = \tau'}{\Gamma; \Delta; \Sigma \vdash e : \tau : \tau'} \quad (E\text{-conv}) \]

Figure 2.9: Expression formation rules (Figure 2.8 continued)
In particular the encoding of the usual oblivious (in proof-passing sense) if is possible using \(B = \lambda t: \text{Bool}. \text{True}\).

Since we allow the definition of higher-kindred recursive types using the \(\mu\) constructor, we need to analyze the structure of \(\kappa\) in the rules E-fold and E-unfold to determine how to unroll the \(\mu\)-type. This analysis is performed by the meta-level macro \(K\) (interpreted by the compiler). The definition of \(K\) is described in detail in Section 2.3.1. Note that the way we define \(K\) will not affect the type safety proof.

The rest expressions have standard meaning.

### 2.2.3 Dynamic Semantics

We present a small step call-by-value operational semantics for CL in the style of Wright and Felleisen [WF94]. The reduction relation \(\Rightarrow\) is defined by the following rules:

\[
\begin{align*}
\text{R-ADD} & & \overline{n_1 + n_2} &\Rightarrow \overline{n_1 + n_2} \\
\text{R-IT-T} & & \overline{n_1} \lessgtr \overline{n_2} &\Rightarrow \overline{n_1} \lessgtr \overline{n_2} \\
\text{R-IT-F} & & \overline{n_1} \geq \overline{n_2} &\Rightarrow \overline{n_1} \geq \overline{n_2} \\
\text{R-FP-T} & & \text{if}[B, A](\text{tt}, X_1, e_1, X_2, e_2) \Rightarrow [A/X_1]e_1 \\
\text{R-FP-F} & & \text{if}[B, A](\text{ff}, X_1, e_1, X_2, e_2) \Rightarrow [A/X_2]e_2 \\
\text{R-\(\beta\)} & & (\lambda x: \tau.e) v \Rightarrow [v/x]e \\
\text{R-TAPP1} & & (\Lambda t: \kappa.e) \tau \Rightarrow [\tau/t]e \\
\text{R-TAPP2} & & (\Lambda X: B.e)[A] \Rightarrow [A/X]e \\
\text{R-FIX} & & \text{fix } x: \tau.f \Rightarrow [\text{fix } x: \tau.f/x]f \\
\text{R-SUB} & & \{l_1 = v_1, \ldots, l_n = v_n\}. l_i \Rightarrow v_i \quad (i \leq n) \\
\text{R-CASE1} & & \text{case}(\text{inj}_1^\tau v, e_1, e_2) \Rightarrow e_1 v \\
\text{R-CASE2} & & \text{case}(\text{inj}_2^\tau v, e_1, e_2) \Rightarrow e_2 v \\
\text{R-UNFD} & & \text{unfold } (\text{fold}^\tau v) \Rightarrow v \\
\text{R-LET} & & \text{let } x = v \text{ in } e \Leftarrow [v/x]e \\
\text{R-TLET1} & & \text{let } t = \tau \text{ in } e \Leftarrow [\tau/t]e \\
\text{R-TLET2} & & \text{let } X = A \text{ in } e \Leftarrow [A/X]e \\
\text{R-OPEN1} & & \text{let } (t, x) = (\tau = \tau', v; \tau') \text{ in } e \Leftarrow [v/x][\tau/t]e \\
\text{R-OPEN2} & & \text{let } (X, x) = (X' = A; v; \tau') \text{ in } e \Leftarrow [v/x][A/X]e
\end{align*}
\]

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

\[
E ::= \bullet \mid E + e \mid v + E \mid E < e \mid v < E \mid E e \mid v E \mid E[\tau] \mid E[A] \\
\mid \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\} \\
\mid E. t \mid \text{inj}_1^\tau E \mid \text{inj}_2^\tau E \mid \text{case}(E, e_1, e_2) \mid \text{fold}^\tau E \mid \text{unfold} E \\
\mid (t = \tau, E; \tau') \mid (X = A, E; \tau) \mid \text{let } x = E \text{ in } e \mid \text{let}(t, x) = E \text{ in } e \\
\mid \text{let}(X, x) = E \text{ in } e \mid \text{if}[B, A](E, X_1.e_1, X_2.e_2)
\]
2.3. EXPRESSIVE POWER OF CL

The notation \( E(e) \) stands for the term obtained by replacing the hole \( \bullet \) in \( E \) by \( e \). The single step computation \( \rightarrow \) relates \( E(e) \) to \( E(e') \) when \( e \leftrightarrow e' \), and \( \rightarrow^* \) is its reflexive-transitive closure.

We can see 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.

2.3 Expressive Power of CL

Based on a powerful type system, our computation language is very expressive. In this section, we will show that by adopting the technique proposed by Collins and Shao [CS02], our CL has a uniform way to support higher-kind recursive types. In addition, the combination of the general recursive type and expressive proof language allows us to support the guarded recursive data types proposed by Xi, Chen and Chen [XCC03].

2.3.1 Higher-Kinded Recursive Types

The standard approach to recursive types in the context of modern programming languages such as SML (as presented by Harper and Stone [HS96]) is to present a higher-kind type constructor \( \mu : (\kappa \rightarrow \kappa) \rightarrow \kappa \). The unrolling of a recursive type \( \mu(\lambda t : \kappa.\tau) \) is then \( \tau[t/\mu(\lambda t : \kappa.\tau)] \). Unfortunately (unless \( \kappa = \Omega \)), the result of applying this type constructor \( \mu \) does not belong to the “ground kind” \( \Omega \), which we define as the kind of all type constructors of terms. In other words, in general we cannot type an expression in the term language with a recursive type. Instead, we would employ Harper/Stone’s solution, which is to use a “projection” operator to take the result back to \( \Omega \). Consider the following simple example (in SML syntax):

```plaintext
datatype tree = Leaf of unit
                | Node of forest * int
and forest = Trees of tree * tree
```

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

\[
\lambda(tree, forest) : \Omega @ \Omega . (unit + forest * int, tree * tree)
\]

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

\[
\tau := \mu(\lambda t : \Omega @ \Omega . (unit + \pi_2 t * int, \pi_1 t * \pi_1 t))
\]

We could then project this result back to obtain the type of `tree` as \( \pi_1 \tau \) and the type of `forest` as \( \pi_2 \tau \).
This approach is not especially general; in particular, in [HS96], projections of $\kappa_1 \rightarrow \kappa_2$ back to $\Omega$ are handled differently in the static semantics from projections of $\kappa_1 \otimes \kappa_2$, and it is not clear how their mechanism may be applied to higher kinds.

\[
\frac{\Delta, t : \kappa \vdash \tau : \kappa}{\Delta \vdash \mu t : \kappa. \tau : \kappa} \quad \text{(rec-form)}
\]

\[
\frac{p = \pi_i \mid \cdot}{s = \tau'' \mid \cdot}
\]

\[
\Delta \vdash \tau \equiv (p(\mu t : \kappa. \tau'))(s) : \Omega
\]

\[
\frac{\Delta : \Sigma \vdash e : \tau}{\Delta ; \Sigma \vdash \text{unfold}(e) : (p(\mu t : \kappa. \tau' / t)[\tau'])(s)} \quad \text{(Unfold)}
\]

\[
\frac{p = \pi_i \mid \cdot}{s = \tau'' \mid \cdot}
\]

\[
\Delta \vdash \tau \equiv (p(\mu t : \kappa. \tau'))(s) : \Omega
\]

\[
\frac{\Delta ; \Sigma \vdash e : (p(\mu t : \kappa. \tau' / t)[\tau'])(s)}{\Delta ; \Sigma \vdash \text{fold}_e(e) : \tau} \quad \text{(Fold)}
\]

Figure 2.10: Typing rules for fold and unfold in Harper and Stone [HS96]

The typing rules given in Figure 2.3.1 are taken from [HS96]. In these rules, $p$ and $s$ are “optional”, and are required to coerce the $\mu$-type back to $\Omega$. The fact that they are necessary indicates this formulation of recursive types is quite unnatural; we have to resort to “tricks” such as this to type check fold and unfold. A worse problem is that these rules only work for a limited set of kinds; i.e. kinds of the form of $(\kappa_1 \otimes \ldots \otimes \kappa_n)$, and for each $\kappa_i$, it must be in the form of $\kappa_i'' \rightarrow \Omega$; it is completely unclear how to extend this mechanism to be more general.

In our design of CL, we use the technique proposed by Collins and Shao [CS02], which explicitly combine the fixpoint function and the projection operation into one type, writing a recursive type over $\kappa$ as $\mu_\kappa(f, g)$. For instance, if we let $f = \lambda t : \Omega \otimes \Omega.(\text{unit} + \pi_2 t * \text{int}, \pi_1 t * \pi_1 t)$ as above, then we can write \text{tree} as

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

and \text{forest} as

$$\mu_{\Theta \Omega}(f, \lambda t : \Omega \otimes \Omega. \pi_2 t).$$

Our typing rules for fold and unfold are given in Figure 2.9 as rules E-fold and E-unfold. The meta-level macro $\mathcal{K}$, interpreted by the compiler, is used to analyse the structure of $\kappa$ in these two rules. Here we extended Collins and Shao’s definition of $\mathcal{K}$ to support kinds $\kappa$ like $\Pi X : A. \kappa'$.\[\]
We define $\mathcal{K}$ as follows (assuming that we have the ability to perform a meta-level typecase operation):

$$\mathcal{K}(\kappa, \tau_1 : \kappa \rightarrow \kappa, \tau_2 : \kappa \rightarrow \Omega) := \tau_2[\mathcal{H}(\kappa, \kappa, \lambda t. t, \tau_1)],$$

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

$$\mathcal{H}(\kappa', \kappa, \tau' : \kappa \rightarrow \kappa', \tau_1 : \kappa \rightarrow \kappa) :=
\begin{cases}
\mu_\kappa(\tau_1, \tau') & \text{when } \kappa' = \Omega \\
(\mathcal{H}(\kappa_1, \kappa, \lambda t. \kappa. \pi_1(\tau'[t]), \tau_1), & \text{when } \kappa' = \kappa_1 \circ \kappa_2 \\
\mathcal{H}(\kappa_2, \kappa, \lambda t. \kappa. \pi_2(\tau'[t]), \tau_1)) & \text{when } \kappa' = \kappa_1 \rightarrow \kappa_2 \\
\lambda X : A. \mathcal{H}(\kappa'', \kappa, \lambda t. (\tau'[t])[X], \tau_1) & \text{when } \kappa' = \Pi X : A. \kappa''
\end{cases}$$

The $\mathcal{K}$ macro takes three arguments: the kind $\kappa$ which we are working over, the fixpoint type function $\tau_1 : \kappa \rightarrow \kappa$, and the coercion type function $\tau_2 : \kappa \rightarrow \Omega$. The macro returns the unrolling of $\mu_\kappa(\tau_1, \tau_2)$. To do this it passes the arguments to the $\mathcal{H}$ macro. The purpose of this macro is to expand the recursive type from something belonging to $\Omega$ to something belong to $\kappa$ so that $\tau_1$ may be applied to it. It takes four parameters: the kind $\kappa'$ which is the kind of the result, the kind $\kappa$ which is the kind of the whole expression (necessary for when we create types $\mu_\kappa(\tau_1, \ldots)$), the type function $\tau_1$ which is copied verbatim to any $\mu$-types we generated, and a type function $\tau'$. The type function $\tau'$ works to take a value from $\kappa$ to $\kappa'$ — we keep this around to build up the new coercion functions when we create $\mu$-types in the recursive calls.

It is easy to check that the definitions of the $\mathcal{K}$ and $\mathcal{H}$ macro are always well-kinded. Also since in each unfolding of the macro $\mathcal{H}$, the first argument $\tau'$ becomes strictly smaller in structure. Therefore the unfolding of the $\mathcal{H}$ macro (and trivially the $\mathcal{K}$ macro) will always terminate. Therefore we say the $\mathcal{K}$ macro is well defined.

Revisiting the example shown at the beginning of this section, we specified the type of $\text{tree}$ to be

$$\mu_{\Omega\Box}(f, \lambda t. \Omega \circ \Omega. \pi_1 t),$$

where

$$f = \lambda t. \Omega \circ \Omega. (\text{unit} + \pi_2 t \ast \text{int}, \pi_1 t \ast \pi_1 t).$$

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

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

Substituting once, this leaves:

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

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

$$\pi_1(f(\mu_{\Omega\Box}(f, \lambda h : \Omega \circ \Omega. \pi_1 h), \mu_{\Omega\Box}(f, \lambda h : \Omega \circ \Omega. \pi_2 h))))$$
which evaluates to
\[
\text{unit} + (\mu_{\Omega\Omega}(f, \lambda x:\Omega@\Omega. \pi_2 x) \times \text{int}),
\]
which is precisely what one would expect.

In [CS02], the authors gave another example to show the formulation of recursive types. Consider the following datatypes (from Olasinski [Oka99]):

\[
\begin{align*}
\text{datatype} & \quad \alpha \text{ Quad} = \quad \Omega \quad \text{of } \alpha \times \alpha \times \alpha \\
\text{and} & \quad \alpha \text{ Square} = \quad \text{Zero of } \alpha \\
& \quad \mid \quad \text{Succ of } (\alpha \text{ Quad}) \text{Square}
\end{align*}
\]

Let \( \kappa = (\Omega \rightarrow \Omega) \oplus (\Omega \rightarrow \Omega) \). We will type \( \alpha \text{ Quad} \) as

\[
\text{Quad} = \lambda t:\Omega. \mu_\kappa(f, \lambda x: \kappa. (\pi_1(x) t)),
\]

and we’ll type \( \alpha \text{ Square} \) as

\[
\text{Square} = \lambda t:\Omega. \mu_\kappa(f, \lambda x: \kappa. (\pi_2(x) t)),
\]

where

\[
f = \lambda x: \kappa. (\lambda t: \Omega. t * t * t, t: \Omega. t + \pi_2(x) (\pi_1(x) t))
\]

In order to create a value of type \( \text{int \quad Quad} \), we would write the following:

\[
\text{fold}^{\text{Quad(int)}}(1, 2, 3, 4).
\]

Similarly, to create the value \( \text{Succ(Zero}(\Omega(1, 2, 3, 4)) \)) of type \( \text{int \quad Square} \), abbreviating \( T = \text{Quad(int)} \):

\[
\text{fold}^{\text{Square(int)}}(\text{inj}_2) \circ \text{Succ}(T)
\]

\[
(\text{fold}^{\text{Square}(T)}(\text{inj}_1) \circ \text{Succ}(\text{Quad}(T)) \circ \text{fold}(T(1, 2, 3, 4))))
\]

### 2.3.2 Guarded Recursive Type Constructors

In Coq or our proof language PL, we can easily define the following inductive data type constructor \( \text{Exp} \):

\[
\begin{align*}
\text{Inductive} \quad & \text{Exp} : \text{Prop} \rightarrow \text{Prop} := \\
& \quad \text{Var} : \Pi \alpha : \text{Prop}. \alpha \rightarrow \text{Exp} \alpha \\
& \quad | \text{Lam} : \Pi \alpha : \text{Prop}. \Pi \beta : \text{Prop}. (\alpha \rightarrow \text{Exp} \beta) \rightarrow \text{Exp}(\alpha \rightarrow \beta) \\
& \quad | \text{App} : \Pi \alpha : \text{Prop}. \Pi \beta : \text{Prop}. \text{Exp}(\alpha \rightarrow \beta) \rightarrow \text{Exp} \alpha \rightarrow \text{Exp} \beta
\end{align*}
\]

However, it is not so clear how to formulate above data type constructor using the general recursive type constructor \( \mu \) and union types.

Xi, Chen and Chen [XCC09] proposed the notion of guarded recursive datatype constructors, which generalized the notion of recursive datatypes and allowed us formulate above inductively defined type constructor using \( \mu \) and union types.

A guarded type is in the form of \( \exists \Delta. \tau \), where \( \Delta \) is a type variable context that may contain some type equalities. The name \textit{guarded} is used for such type
equalities. For instance, $\exists \Delta_1.\tau$ is a guarded type, where $\Delta_1 = \{\alpha_1, \alpha_2, \alpha_1 \ast \alpha_2 \equiv \text{int} \ast \text{bool}\}$ and $\tau = \alpha_1 \ast \alpha_1$; this type is equivalent to $\text{int} \ast \text{int}$ since we must map $\alpha_1$ to $\text{int}$ in order to satisfy the type equality $\alpha_1 \ast \alpha_2 \equiv \text{int} \ast \text{bool}$. The type $\exists \Delta_2.\tau$ is also a guarded type, where $\Delta_2 = \{\alpha_1, \alpha_2, \alpha_1 \ast \alpha_2 \equiv \text{int}\}$; this type is equivalent to the type void, i.e., the type in which there is no element, since the type equality $\alpha_1 \ast \alpha_2 \equiv \text{int}$ cannot be satisfied.

For a recursive datatype constructor in the following general form (in a SML-like syntax):

$$\text{datatype } \vec{\alpha} \ T = \begin{cases} c_1 \text{ of } \forall \alpha_1' : \kappa_1'. \tau_1 \rightarrow \tau_1' \ T \\ c_2 \text{ of } \forall \alpha_2' : \kappa_2'. \tau_2 \rightarrow \tau_2' \ T \\ \vdots \\ c_k \text{ of } \forall \alpha_k' : \kappa_k'. \tau_k \rightarrow \tau_k' \ T \end{cases}$$

it can be formally defined as $\mu \text{.}\sigma$, where $\sigma$ is a union of guarded types:

$$\mu \text{.}\lambda \vec{\alpha}. (\exists \alpha_1, \tau_1 \equiv \vec{\alpha}).[t/T]\tau_1 + \ldots + \exists \alpha_k, \tau_k \equiv \vec{\alpha}).[t/T]\tau_k)$$

That is why $T$ is called a guarded recursive datatype constructor (it is assumed that for $i = 1, \ldots, k$, $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ and $\vec{\alpha}_i$ share no common type variables).

In [XCC03] the authors gave rules to infer and solve the type constraints automatically so that programmer do not have to explicitly write these type equations. Although in our type system, we do not have similar mechanics that allow user explicitly specify the type constraints, nor do we have such constraint-solving rules, we can support such guarded recursive type constructors if the type variables $\vec{\alpha}$ have kinds $\vec{A}$, i.e. they are type variables in the proof language.

The basic idea is that we can inductively define the equality of types in our PL:

$$\text{Inductive Eq}[k : \text{Prop}, t : k] : k \rightarrow \text{Prop} := \text{refl}\_\text{equal} : \text{Eq} k t t;$$

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

$$\text{Leibniz : } \Pi k : \text{Prop}. \Pi t : k. \Pi t' : k.(\text{Eq} k t t') \rightarrow \Pi : (k \rightarrow \text{Prop}). P \ t \rightarrow P \ t'$$

Now let’s consider the classical example that use dependent types to eliminate index bound checking [X98] (in DML syntax):

$$\text{datatype } \vec{\alpha}\ 1\text{\ list with nat } = \begin{cases} \text{nil}(0) \\
\{n : \text{nat}\} \text{ cons}(n + 1) \text{ of } \vec{\alpha} * \vec{\alpha}\ 1\text{\ list}(n) \end{cases}$$

The syntax introduces a type constructor $\text{list}$ that takes a type and a index of sort $\text{nat}$ to form a list type.

Above type constructor can be easily defined in Vero:

$$\text{list } = \Lambda t : \Omega. \Lambda X : \text{Nat}. \mu \Omega. \text{Nat} \rightarrow \Omega (f, g),$$

where $f$ is defined as:

$$f = \Lambda S : \Omega \rightarrow \text{Nat} \rightarrow \Omega. \Lambda t : \Omega. \Lambda X : \text{Nat}.\ (\exists p_1 : (\text{Eq Nat} \ X \ \text{zero}). \text{unit} + \exists X' : \text{Nat}. \exists p_2 : (\text{Eq Nat} \ X \ (\text{succ} \ X')). (t \ast S[t][X']););$$
and \( g \) is:

\[
g = \Lambda f : \Omega \rightarrow \text{Nat} \rightarrow \Omega. f[t][X].
\]

As we can see, \( p_1 \) and \( p_2 \) in \( f \) are actually proofs of formulae \((\text{Eq Nat } X \text{ zero})\) and \((\text{Eq Nat } X (\text{succ } X'))\), respectively. This kind of guarded recursive type constructors are of special interests to us because it allows us package the proof terms and computation terms together. Later on we can extract the proof terms by unpacking such a package. For instance, in the following program, we can unpack \( x_1 \) in \( e_1 \) and \( x_2 \) in \( e_2 \) to get the proof term \( p_1 \) and \( p_2 \) respectively.

\[
\begin{align*}
\text{let} & \\
& \quad t : \Omega \quad = \ldots \\
& \quad X : \text{Nat} \quad = \ldots \\
& \quad l : \text{list}[t][X] = \ldots \\
\text{in} & \\
\text{case(unfold } l, \\
& \quad \lambda x_1 : (\exists p_1 : (\text{Eq Nat } X \text{ zero}).\text{unit}).e_1, \\
& \quad \lambda x_2 : (\exists X' : \text{Nat}. \exists p_2 : (\text{Eq Nat } X (\text{succ } X')).(t * \text{list}[t][X'])).e_2)
\end{align*}
\]

Another important usage of the notion of guarded recursive type constructor is to construct singleton type constructors, like \( s_{\text{Nat}} \) and \( s_{\text{Bool}} \). In fact, the primitive singleton type constructor \( s_{\text{Bool}} \) can be defined in the following way:

\[
s_{\text{Bool}} = \Lambda X : \text{Bool}.(\exists p_1 : (\text{Eq Bool } X \text{ true}).\text{unit}) + (\exists p_2 : (\text{Eq Bool } X \text{ false}).\text{unit})
\]

The if expression can now be simulated by the more general case expression:

\[
\begin{align*}
\text{let} & \\
& \quad b : s_{\text{Bool}}[M] = \ldots \\
& \quad X : \Pi X' : \text{Bool}.B X' = AX'.B \ldots \\
& \quad X' : B M = X M \\
\text{in} & \\
\text{case}(b, \\
& \quad \lambda x_1 : (\exists p_1 : (\text{Eq Bool } M \text{ true}).\text{unit}). \\
& \quad \text{let} \\
& \quad \quad (p_1, _) = x_1 \\
& \quad \quad X_1 = \text{Leibniz Bool } M \text{ true } p_1 \ B \ X' \\
& \quad \quad \text{in } e_1, \\
& \quad \lambda x_2 : (\exists p_2 : (\text{Eq Bool } M \text{ false}).\text{unit}). \\
& \quad \text{let} \\
& \quad \quad (p_2, _) = x_2 \\
& \quad \quad X_2 = \text{Leibniz Bool } M \text{ false } p_2 \ B \ X' \\
& \quad \quad \text{in } e_2)
\end{align*}
\]

where the \( B, X', b, e_1 \) and \( e_2 \) above are corresponding to the \( B, A, e_0, e_1 \) and \( e_2 \) in the if expression, respectively.
Note that in general, it is undecidable to automatically infer such proof terms as $p_1$ and $p_2$. That is why in DML [Xi98] only a very constraint domain is used in the position of $p_1$ and $p_2$. However, since Vero is an explicitly typed language, we require the programmer to provide such proof terms to construct a value of such a list type. Therefore our language is much more expressive than Hongwei Xi’s guarded recursive datatype constructors [XCC03] or DML [Xi98].

We define another singleton type constructor, i.e. singleton list type constructor. First we define the List constructor in our PL:

\[
\text{Inductive List : Prop := Nil : List} \\
\text{\quad | Cons : Nat \to List \to List.}
\]

The singleton list constructor in the computation language is defined as:

\[
s_{\text{List}} : \Pi X : \text{List}. \Omega : \text{List}. \mu \Pi X \text{List.} \Omega(f, g),
\]

where $f$ and $g$ are defined as following:

\[
f = \Lambda t : (\Pi X : \text{List.} \Omega). \Lambda X : \text{List.} \\
\quad \exists p_1 : (\text{Eq List X Nil}). \text{unit +} \\
\quad \exists n : \text{Nat.} \exists l : \text{List.} \exists p_2 : (\text{Eq List X (Cons a l))}. (s_{\text{Nat}}[n] \ast t[l])
\]

\[
g = \Lambda t : (\Pi X : \text{List.} \Omega). (t[X])
\]

Now we can define an empty list:

\[
\text{nil} : s_{\text{List}}[\text{Nil}] = \text{fold}^{\text{unit}}_{\text{Nil}} \langle \text{inj}_1 \rangle \langle p_1 = \text{Pr} (\text{Eq List Nil Nil}), () : \text{unit} \rangle
\]

where $\text{Pr}(\text{Eq List Nil Nil})$ stands for a proof of the formula $\langle \text{Eq List Nil Nil} \rangle$, which is trivial and omitted here. For brevity we also omit the superscript of $\text{inj}_1$.

We can give another non-empty instance of the singleton list defined above, e.g.,

\[
l : s_{\text{List}}[\text{Cons zero Nil}] = \text{fold}^{\text{unit}}_{\text{Cons zero Nil}} \langle \text{inj}_2 \rangle \\
\quad \langle a = \text{zero}, l = \text{Nil}, \\
\quad \langle p_2 = \text{Pr} (\text{Eq List (Cons zero Nil) (Cons a l))} \rangle \langle 0, \text{nil} : s_{\text{Nat}}[n] \ast (s_{\text{List}} l) \rangle \\
\quad : \exists p_2 : (\text{Eq List (Cons zero Nil) (Cons a l))}. (s_{\text{Nat}}[a] \ast (s_{\text{List}} l)) \\
\quad : \exists l : \text{List.} \exists p_2 : (\text{Eq List (Cons zero Nil) (Cons a l))}. (s_{\text{Nat}}[a] \ast (s_{\text{List}} l))).
\]

Again the proof of the formula $\langle \text{Eq List (Cons zero Nil) (Cons a l)} \rangle$ is omitted here, which is denoted as $\text{Pr}(\text{Eq List (Cons zero Nil) (Cons a l)})$. The superscript of $\text{inj}_2$ is also omitted.

**Automatically generating the indexed type constructor.** From above examples we can see that if the programmer wants to define both an inductive type in PL and its counterpart (i.e. the type constructor indexed over such an inductive PL type) in CL, he/she need do a lot of repeated work, because the similarity of these two in structures.
We also noticed that there are some patterns in the definitions of singleton type constructors like \(s_{\text{bool}}\) and \(s_{\text{list}}\). It is likely that there is a general way that, given an inductive PL type, can automatically generate its counterpart in CL. Here we present our attempt to develop such a general method.

Before presenting such a method, we first have a review of the syntactic constraints of types of constructors in inductive definition [Pau93].

**Definition 2.13** The terms \(P\) which are strictly positive with regard to \(X\) are generated by the syntax:

\[
\text{Pos} ::= X | \text{Pos} \; m | \Pi x: M. \text{Pos}
\]

with the restriction that \(X\) does not occur free in \(M\) and \(m\).

**Definition 2.14** The terms \(C\) which are a type of the constructor with regard to \(X\) are generated by the syntax:

\[
\text{Co} ::= X | \text{Co} \; m | P \rightarrow \text{Co} | \Pi x: M. \text{Co}
\]

with the restriction that \(P\) is strictly positive with regard to \(X\), and \(X\) does not occur free in \(M\) and \(m\). Also we require that \(C\) is a well kinded type.

A well-formed constructor type can be written as \(C \equiv \Pi X_i : A_i . X.B_i\). A strictly positive type is a particular case of constructor types, which we call a non-recursive type of constructor. A type of constructor which is not a strictly positive type is called recursive. Note that in above definition the dependent type \(\Pi x : M. \text{Co}\) and non dependent type \(P \rightarrow \text{Co}\). \(X\) can occur in the type of the argument only in the non dependent case.

In the following discussion, we will relax the formulation in Section 2.1. We say a general inductive definition is in the form of \(\text{Ind}(X_i : A_i | X : \Pi Y_i : B_i.s) | C_i\). This general form allow inductive definitions to take arguments \(X_i\), as in Coq [Tea09]. The kind of the inductive type \(X\) is a general arity \(\Pi Y_i : B_i.s\), where \(s\) is Prop or Type. A restricted inductive definition is in the form of \(\text{Ind}(X : \text{Prop}) | C_i\), which takes no arguments and requires the kind of \(X\) must be Prop. It has the further restriction that in each \(C_i\), which is in the form of \(\Pi X_i : A_i . X\), each \(A_i\) must also be an inductive type defined using the restricted form.

First we work on the restricted form of inductive definition. Our goal is to define a meta-level macro \(S\), which takes a restricted inductive definition and generates the corresponding indexed type constructor in CL. If the inductive PL type \(I\) has recursive constructors, \(S\) will generate the CL type constructor \(\Lambda X : I.(\mu \Pi X : I. \Omega(f, g))\), where

\[
f : (\Pi X : I. \Omega) \rightarrow \Pi X : I. \Omega = \Lambda t : (\Pi X : I. \Omega). \Lambda X : I. (\tau_1 + \ldots + \tau_k)
\]

and

\[
g : (\Pi X : I. \Omega) \rightarrow \Omega = \Lambda t : (\Pi X : I. \Omega). (t[X]).
\]
Each $\tau_i$ is generated according to the constructor type $C_i$ in $I$. If all the constructors of $I$ are non-recursive, $S$ will simply generate a type constructor

$$t : \Pi X : I. \Omega = \Lambda X : I. (\tau_1 + \ldots + \tau_k),$$

where $\tau_i$ has the same meaning with above definition of $f$.

The central task of $S$ is to generate $\tau_i$ for each constructor type

$$C_i = \Pi X : A_i. I.$$

Recall that in the restricted form we require each $A_i$ must be an inductive type which also has the restricted form. $S$ generates $\tau_i$ in the following way:

$$\tau_i = \exists X_i : A_i. \exists p : (\text{Eq } I X (c_i X_i)). ((S' A_1)[X_1] * \ldots * (S' A_n)[X_n]),$$

where

$$S' A_i = \begin{cases} S A_i & A_i \not= I \\ t & A_i = I \end{cases}$$

The implementation of $S$ is given in pseudo SML code in Figure 2.11.

```sml
fun S (I as Ind(X : Prop)[C : G]) = 
  if I is recursive, then
    \mu I.X.\Omega(\Lambda t : (\Pi X : I.\Omega A X : I. (S C (t, I, c_1, C_1) + \ldots + S C (t, I, c_n, C_n)), g))
  else A X : I.(S C (I, I, c_1, C_1) + \ldots + S C (I, I, c_n, C_n))
  and S C (t, I, c_i, C_i) =
    let fun F I [] [] = \exists p : (\text{Eq } I X c_i).unit
      F I [(X_1 : A_1), \ldots, (X_k : A_k)] [t_1, \ldots, t_k] =
        \exists X_i : A_i. \exists p : (\text{Eq } I X (c_i X_i)).(t_1 * \ldots * t_k)
    F (\Pi X : A. C) L_1 L_2 = F C (L_1 \circ ([X : A])) (L_2 \circ ([S A][X]))
    F (I \rightarrow C) L_1 L_2 = F C (L_1 \circ ([X : I])) (L_2 \circ [t[X]])
    F (A \rightarrow C) L_1 L_2 = F C (L_1 \circ ([X : A])) (L_2 \circ ([S A][X]))
  in
    F C [ ] [ ]
  end
```

Figure 2.11: Implementation of $S$.

However, above definition of $S$ cannot be applied to the general form of inductive definition for two reasons: first, $S$ is at meta-level and it could not be defined using the type language, therefore we have no way to deal with $S X$ where $X$ is a variable in PL; second, in general we have no idea about how to
handle a PL term which is not an inductive type, e.g., the arrow term \( A \to B \), in other words, if the \( A_i \) shown in the definition of \( S' \) \( A_i \) is a general PL form, we do not know how to do the conversion.

One way to relax such restrictions is not to convert any \( A_i \) which is different with \( I \). This approach will, of course, generate a different kind of indexed type constructors.

Recall that a *general* inductive type \( I \) has the form \( \text{Ind}[X_i : A_i](X : \Pi Y_i : B_i, s)[\xi_i : \xi_i] \). We define a new macro \( S_I \) to handle the general form.

For the recursive case, we generate the following CL type constructor (abbreviating \( T = \Pi X_i : A_i, \Pi Y_i : B_i, \Pi X : (I X_i, Y_i), \Omega \)):

\[
\Lambda X_i : A_i, \Lambda Y_i : B_i, \Lambda X : (I X_i, Y_i), \mu_T(f, g),
\]

where

\[
f : T \to T = \Lambda t : T. \Lambda X_i : A_i, \Lambda Y_i : B_i, \Lambda X : (I X_i, Y_i), (\tau_1 + \ldots + \tau_k),
\]

and

\[
g : T \to \Omega = \Lambda t : T. (t[X_i][Y_i][X]).
\]

Again each \( \tau_i \) is generated according to the constructor type \( C_i \) in \( I \). If all the constructors of \( I \) are non-recursive, \( S_I \) will simply generate a type constructor

\[
t : T = \Lambda X_i : A_i, \Lambda Y_i : B_i, \Lambda X : (I X_i, Y_i), (\tau_1 + \ldots + \tau_k).
\]

For a general constructor type (note the \( \tilde{A}_i \) and \( \tilde{B}_i \) is different with the \( A_i \) and \( B_i \) in \( I \)):

\[
C_i = \Pi X_i : A_i, I \tilde{B}_i,
\]

the form of \( \tau_i \) is also different with that for the restricted form. For non-recursive case, we simply generate:

\[
\tau_i = \exists X_i : A_i \exists p : (\text{Eq} (I X_i, Y_i) X (c_i X_i X_i^p)). \text{unit};
\]

while for recursive case,

\[
\tau_i = \exists X_i : A_i \exists p : (\text{Eq} (I X_i, Y_i) X (c_i X_i X_i^p))((S'_i A'_i) \times \cdots \times (S'_i A'_n) \times X_i^p),
\]

where

\[
(S'_i A'_i) [X_i] = \begin{cases} \text{unit} & A'_i \neq I \tilde{Y}_i^p \text{ for any } \tilde{Y}_i^p \\ t[X_i] & \exists Y_i^p \text{ such that } A'_i = I Y_i^p \end{cases}
\]

Note the arguments \( X_i \) are the same for all appearance of \( I \), therefore they are omitted. This agrees with Coq syntax.

Then we do some clean-up: drop unit in the tuple type and only keep these \( t[X_i] \) for any \( i \), resulting in some \( \tau_i \) like:

\[
\tau_i = \exists X_i : A_i \exists p : (\text{Eq} (I X_i, Y_i) X (c_i X_i X_i^p))((t[X_i^p] \ast \ldots \ast t[X_i^n]),
\]

\]}
2.3. EXPRESSIVE POWER OF CL

Applying above approach to the following polymorphic list:

\[ \text{Inductive List}[X:\text{Prop}] : \text{Prop} := \text{Nil} : \text{List} \]
\[ \mid \text{Cons} : X \rightarrow \text{List} \rightarrow \text{List}, \]

we get the following indexed type constructor in CL (abbreviating \( T = \exists X' : \text{Prop}.\exists X : (\text{List} X'').\Omega)\):

\[ \Lambda X' : \text{Prop}.\Lambda X : (\text{List} X').\mu_T(f, g), \]

where \( f \) and \( g \) are defined as following:

\[ f = \lambda t : T.\Lambda X' : \text{Prop}.\Lambda X : \text{List} X'. \]
\[ \exists p_1 : \text{(Eq (List X') X (Nil X')).unit} + \]
\[ \exists x : X'. \exists : (\text{List} X').\exists p_2 : (\text{Eq (List X')} X (\text{Cons} X' x l)).t[l] \]

\[ g = \lambda t : T.t[X'][X] \]

However, such an indexed type constructor might not be so interesting from the viewpoint of computation: after type ensure, only an empty list is left. We have no idea how useful it is in practice.

Maybe in the future we can come up with some practical approach, which is a compromise between the restricted \( S \) and the general but not so interesting \( S_t \). Let’s call it \( S_G \). \( S_G \) works in the same way with \( S_t \) except it handles the general constructor type

\[ C_i = \exists X_i : \tilde{A}_i. I \tilde{B}_i' \]

in a different way, i.e. it will generate a different \( \tau_i \):

\[ \tau_i = \exists \tilde{X}_i : \tilde{A}_i. \exists p : \text{(Eq (I \tilde{X}_i \tilde{Y}_i) X (c_i \tilde{X}_i \tilde{X}_i'))}.((S'_G A'_i)[X'_i] * \cdots * (S'_G A'_n)[X'_n]), \]

where

\[ (S'_G A'_i)[X'_i] = \begin{cases} 
(S_{A'_i} A'_i)[X'_i] & \text{A'}_i \neq I \tilde{Y}_i, \text{but programmer provides such} \ S_{A'_i} \text{for} \ A'_i \\
\text{unit} & \text{A'}_i \neq I \tilde{Y}_i \text{and no} \ S_{A'_i} \\
t[X_i] & \text{exists} \ Y_i \text{such that} \ A'_i \equiv I \tilde{Y}_i 
\end{cases} \]

Then we do the same clean-up as \( S_t \) to remove (or merge) useless unit type.

Of course to make such an approach really work, we must provide some way to allow the programmer to point out which \( A'_i \) he/she wants to convert, and to provide corresponding \( S_{A'_i} \). This will be part of our future work.
CHAPTER 2. THE INTERNAL LANGUAGE
Chapter 3

The Vero Surface Language

3.1 Lexical Conventions

In this section we define the Vero lexical convention, which is very close to that of SML.

3.1.1 Notation Conventions

In the following definition of lexical objects, we adopt the following notation conventions:

\[ \langle obj \rangle \quad \text{the lexical object } obj \text{ appears zero or one time} \]

\[ \{ obj \}^* \quad \text{the lexical object } obj \text{ appears zero or more times} \]

\[ \{ obj \}^+ \quad \text{the lexical object } obj \text{ appears one or more times} \]

Note that \( \langle \) and \( \rangle \) are different with \( < \) and \( > \) respectively. The latter belong to the alphabet of Vero, while the former only appear in production rules.

3.1.2 White Space

Space, newline and horizontal tabulation are considered as white spaces. White spaces are ignored but they separate tokens.

3.1.3 Comments

Comments in Vero are enclosed between \( \ast \) and \( \ast \), and can be nested. Comments are treated as white spaces.

3.1.4 Identifiers and Operators

At present we only define the core language without the module system, therefore we do not define qualified identifiers here.
An identifier is a sequence of *letters, digits, primes* (*) and underbars (*_*), starting with a letter or prime. An operator is a *symbolic* sequence, i.e. any non-empty sequence of *symbols*.

\[
\begin{align*}
\text{non zero} &::= 1..9 \\
\text{digit} &::= 0 \mid \text{nonzero} \\
\text{letter} &::= A..Z \mid a..z \\
\text{symbol} &::= ! \mid \% \mid \& \mid \$ \mid \# \mid + \mid - \mid / \mid : \mid < \\
&\quad \mid = \mid > \mid ? \mid @ \mid ^ \mid _ \mid \` \mid | \mid \^ \mid \_* \\
\text{id} &::= \text{letter} \{ \text{digit} \mid \text{letter} \mid _ \mid ? \}_* \\
\text{id} &::= \{ \text{symbol} \}_+ 
\end{align*}
\]

In either case, however, reserved words are excluded. This means that for example # and | are not operators, but ## and |=| are.

Identifiers are case-sensitive.

All operators by default are infix. In section 3.2 we will see how to convert the infix operator to a prefix function identifier.

### 3.1.5 Literals

#### Numeral Literals

Vero supports both decimal and hexadecimal numerals.

\[
\begin{align*}
\text{hexdigit} &::= \text{digit} \mid A..F \mid a..f \\
\text{decInt} &::= (\^) \{ \text{digit} \}_+ \\
\text{hexInt} &::= (\^) 0x \{ \text{hexdigit} \}_+ \\
\text{int} &::= \text{decInt} \mid \text{hexInt} \\
\text{word} &::= 0w \{ \text{digit} \}_+ \mid 0wx \{ \text{hexdigit} \}_+ 
\end{align*}
\]

At present we do not support *real* numbers.

#### String Literals

The *Vero alphabet* includes the set of printable ASCII characters (i.e., numbered 33-126), space or escape sequences (see the following sub-section). Unicode is not supported.

A string literal is a sequence, between quotes (*"*), of zero or more characters in the alphabet.

\[
\text{string} ::= " \{ \text{alphabet} \} " 
\]
3.1. LEXICAL CONVENTIONS

Character Literals
A character literal is a sequence of the form \#s, where s is a string constant denoting a string of size one character.

\textit{char} ::= \#"alphabet"

Escape Sequence for Character and String Literals
The character and string \textit{escape sequences} allow for the representation of some non-graphic characters as well as the single quote, double quote, and backslash characters in character literals and string literals.

Each escape sequence starts with the escape character \\ (backslash). The escape sequences are:

- \a A single character interpreted by the system as alert (ASCII 7)
- \b Backspace (ASCII 8)
- \t Horizontal tab (ASCII 9)
- \n Linefeed, also known as newline (ASCII 10)
- \v Vertical tab (ASCII 11)
- \f Form feed (ASCII 12)
- \r Carriage return (ASCII 13)
- \\c The control character c, where c may be any character with number 64-95. The number of \\c is 64 higher than c.
- \ddd The single character with number ddd (3 decimal digits denoting an integer in the ordinal range of the alphabet).
- \uxxxx The single character with number xxxx (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
- \\\\ " This sequence is ignored, where f..f stands for a sequence of one or more formatting characters.

The \textit{formatting characters} are a subset of the non-printable characters including at least space, tab, newline, form feed. The last form allows long strings to be written on more than one line, by writing \\ at the end of one line and at the start of the next.

Reserved words
The following words are reserved keywords in Vero (incomplete):

\textit{resvWords} ::= [1 | 1 | 1 | # | ; | . | ... | : | :: | := | = | # | -> |
| -> | * | < | > | and | Cases | case | cond | else | end
| End | Fix | fn | if | in | Inductive | let | of | Pack
| rec | then | type | val | with | andalso | orelse
3.2 Syntax

In this section, we give the grammar of the Vero surface language. We show some sample Vero programs in Section 3.3.

In Vero identifiers and operators have distinguished lexical form, as described in Section 3.1.4. Programmers can introduce new operators, which must be infix and have lower priority than built-in operators. We do not provide directives for fixity and priority declaration. In the definition of a new operator opr, opr must be in parentheses and defined as a prefix function identifier (i.e. in the form of (opr), see Figure 3.1). After definition, opr can be used as a infix operator, or a prefix function identifier if it is bracketed in parentheses.

\[
pre_id ::= id \mid (opr)
\]
\[
op ::= opr \mid andalso \mid orelse \mid + \mid - \mid * \mid / \mid \text{div}
\mid \text{mod} \mid < \mid <= \mid > \mid >=
\]

Figure 3.1: Identifiers

Figure 3.2 shows the grammar of the type language of Vero. Figures 3.4 and 3.5 define the grammar of Vero declarations and expressions. A Vero program is either a declaration or an expression, as defined in Figure 3.6.

\[
ty ::= app_ty \mid \{tlabels\} \mid ty \cdot ty \mid ty \rightarrow ty \mid ty \ opr \ ty \mid ty \ opr \ ty \mid ty \\
\mid \{cids\} \ ty \mid \{cids\} \ ty
\]
\[
app_ty ::= aty \mid aty \ app_ty
\]
\[
aty ::= pre_id \mid \{tlabels\} \mid \{union\_brches\} \mid \{t\}
\mid \{t\} :: Cases \ ty \ of \ prrules \ End
\]
\[
tlabels ::= \epsilon \mid tlabels'
\]
\[
tlabels' ::= selector : ty \ , \ tlabels' \mid selector : ty
\]
\[
union\_brches ::= union\_brch \mid union\_brch \mid union\_brches
\]
\[
union\_brch ::= id \mid pre\_id \ of \ ty
\]
\[
prrules ::= prmatch \mid prmatch \mid prrules
\]
\[
prmatch ::= simp\_pat => ty
\]

Figure 3.2: The Type Language
simp\_pat ::= sgl\_pat \mid pre\_id \textsc{spats}

sgl\_pat ::= pre\_id \mid \_  

\textsc{spats} ::= sgl\_pat \mid sgl\_pat \textsc{spats}

Figure 3.3: Simple Pattern

dec ::= \texttt{val \_} \texttt{vb} \mid \texttt{val \_} \texttt{rec \_} \texttt{rvbs} \mid \texttt{type \_} \texttt{tb} \mid \texttt{type \_} \texttt{rec \_} \texttt{rtbs} \mid \texttt{open \_} \texttt{opn}
| \texttt{Inductive \_} \texttt{tyfarg \_} \texttt{indlist} \mid \texttt{Fix \_} \texttt{tyfbs}
decs ::= \varepsilon \mid \textsc{dec \_} \textsc{decs} \mid ; \textsc{decs}
vb ::= \texttt{pre\_id} = \texttt{exp} \mid \texttt{pre\_id} : \texttt{ty} = \texttt{exp}
\texttt{rvb} ::= \texttt{pre\_id} : \texttt{ty} = \texttt{exp}
\texttt{rvbs} ::= \texttt{rvb} \mid \texttt{rvb} \texttt{and} \texttt{rvbs}
tb ::= \texttt{pre\_id} = \texttt{ty} \mid \texttt{pre\_id} : \texttt{ty} = \texttt{ty}
\texttt{rtb} ::= \texttt{pre\_id} : \texttt{ty} = \texttt{ty}
\texttt{rtbs} ::= \texttt{rtb} \mid \texttt{rtb} \texttt{and} \texttt{rtbs}
\texttt{opn} ::= <\texttt{id},\texttt{id}> = \texttt{exp}
\texttt{tyfarg} ::= \varepsilon \mid \texttt{tyfarg}'
\texttt{tyfarg}' ::= [ \texttt{cids} ]
cids ::= \texttt{cid} \mid \texttt{cid} , \texttt{cids}
cid ::= \texttt{id} : \texttt{ty}
\texttt{indlist} ::= \texttt{inditem} \mid \texttt{inditem \_} \texttt{with \_} \texttt{indlist}
\texttt{inditem} ::= \texttt{pre\_id} : \texttt{ty} = \texttt{constructors}
\texttt{constructors} ::= \varepsilon \mid \texttt{constructors}'
\texttt{constructors}' ::= \texttt{constructor} \mid \texttt{constructor} \mid \texttt{constructors}'
\texttt{constructor} ::= \texttt{pre\_id} : \texttt{ty}
\texttt{tyfbs} ::= \texttt{tyfb} \mid \texttt{tyfb} \texttt{with} \texttt{tyfbs}
\texttt{tyfb} ::= \texttt{pre\_id} \texttt{tyfarg} : \texttt{ty} = \texttt{ty}

Figure 3.4: Declaration
exp ::= app_exp | tyfarg' exp | # selector aexp | exp op exp
    | if exp then exp else exp | case exp of match
    | fn earg_pat => exp
app_exp ::= aexp | aexp app_exp
aexp ::= pre_id | int | word | string | char | ( seqexp ) | ()
    | ( exp_2c ) | ( elabels ) | [[ exp_list ]] | #[exp_list]
    | let decs in seqexp end | aexp tyarg | id [ ty ]
    | cond [ ty , ty ] ( condbody ) | Pack id = ty with exp : ty End
seqexp ::= exp | exp ; seqexp
exp_2c ::= exp, exp_2c | exp, exp
elabel ::= selector = exp
elabels' ::= elabel , elabels' | elabel
elabels ::= ε | elabels'
exp_list' ::= exp | exp , exp_list'
exp_list ::= ε | exp_list'
tyarg ::= [ ty ]
condbody ::= exp , [ id ] exp , [ id ] exp
selector ::= id | int
match ::= rule | rule | match
rule ::= simp_pat => exp
earg_pat ::= cid | ( cids )

Figure 3.5: Expressions

program ::= dec | exp

Figure 3.6: Program
3.3 Sample Vero Programs

To show the meaning of Vero grammars, we present some sample programs.

3.3.1 Proof Terms

In Figure 3.7 we give the representation of the PL terms defined in Figure 2.2 (Note that in Figure 3.7 we changed the name of the function Cond in Figure 2.2 to \textit{Test}).

As we can see the syntax of PL terms is quite similar with Coq [Tea03]. One difference is that we have no type inference in Vero, therefore the result type of \textit{Cases} expression must always be explicitly provided, even if it is non dependent.

Figures 3.8 and 3.9 shows more PL terms, which includes \textit{List}, \textit{Eq}, \textit{Leibniz} and other terms defined in Section 2.3.2 and in the paper on TSCB [SSTP02].

In our surface language, we support mutually inductive types and mutually recursive fixpoints, as in Coq. The following program is one example of mutually recursive fixpoints.

\begin{verbatim}
Fix even[n: Nat] : Bool =
  <: Bool :> Cases n of 0 => True
         | S n' => odd n'
End

with odd[n: Nat] : Bool =
  <: Bool :> Cases n of 0 => False
         | S n' => even n'
End
\end{verbatim}
(**************************
(* some built-in PL terms *)
**************************)

(* Syntax of Inductive definition and Cases expression is *)
(* almost the same with Coq (v.7.4) *)
Inductive Nat : Prop = 0 : Nat | S : Nat -> Nat;

Inductive Bool : Prop = True : Bool | False : Bool;

(* Fix here corresponds to the Fixpoint command in Coq *)
Fix plus [n1: Nat]: Nat -> Nat =
  [n2: Nat] <:: Nat -> Cases n1 of 0 => n2
  | S n' => S (plus n' n2)
End;

(* type is like the Definition command in Coq Vernacular *)
(* the constraint after the identifier ifez is optional *)
type ifez : Nat -> (k:Prop)k -> (Nat -> k) -> k =
  [n: Nat][k: Prop, p1:k, p2:Nat->k]
  <:: k -> Cases n of 0 => p1
  | S n' => p2 n'
End;

Fix le [n1: Nat] : Nat -> Bool =
  [n2: Nat]
  <:: Bool -> Cases n1 of 0 => True
  | S n' => ifez n2 Bool False (le n')
End;

type lt : Nat -> Nat -> Bool = [n1: Nat, n2 : Nat] le (S n1) n2;

type Test : Bool -> Prop -> Prop -> Prop =
  [b:Bool, p1:Prop, p2:Prop]
  <:: Prop -> Cases b of True => p1
  | False => p2
End;

Figure 3.7: Representation of some built-in PL terms
3.3. SAMPLE VERO PROGRAMS

Inductive [a:Prop] List : Prop =
    Nil : List
 | Cons : a -> List -> List;

Inductive [k:Prop, x:k] Eq : k -> Prop = refl_equal : Eq x;

type LeibKnd =
    (k : Prop, x: k)
    (P: k -> Prop) (P x) -> (y:k) (Eq k x y) -> (P y);

type Leibniz : LeibKnd =
    [k:Prop, x:k, P: k->Prop, p1: P x, y:k, p2: Eq k x y]
    <: P : Cases p2 of refl_equal => p1 End;

Inductive LT : Nat -> Nat -> Prop =
    ltzs : (n:Nat) LT 0 (S n)
 | ltss : (n1:Nat, n2:Nat) LT n1 n2 -> LT (S n1) (S n2);

type TT : Prop = (P : Prop) P -> P;

type id : TT = [P:Prop, p:P] p;

type LTUrTrue : Nat -> Nat -> Bool -> Prop =
    [n1:Nat, n2:Nat, b:Bool] Test b (LT n1 n2) TT;

type LTCond = [n1: Nat, n2: Nat] LTUrTrue n1 n2 (it n1 n2);

type LTsucc =
    [n1: Nat, n2: Nat, b:Bool]
    LTUrTrue n1 n2 b -> LTUrTrue (S n1) (S n2) b;

Fix ltPrf [n1:Nat] : (n2:Nat) LTCond n1 n2 = [n2: Nat]
<: [n:Nat] LTCond n n2 :
    Cases n1 of
    0 => <: LTCond 0 :
    Cases n2 of
    0 => id
    | S n2' => (ltzs n2')
End
| S n1' => <: LTCond (S n1') :
    Cases n2 of
    0 => id
    | S n2' =>
        ( <: LTsucc n1' n2' :
            Cases (lt n1' n2') of
            True => ltss n1' n2'
            | False => id TT
            End) (ltPrf n1' n2')
    End
End

Figure 3.8: More PL terms
type leftUnit : (n:Nat) Eq Nat n (plus 0 n) =
       [n:Nat] refl_equal Nat n;

type eqf : (k1: Prop, k2:Prop, f: k1 -> k2)
       (t1: k1, t2:k1, p: Eq k1 t1 t2) Eq k2 (f t1) (f t2) =
       [k1: Prop, k2:Prop, f: k1 -> k2] [t1: k1, t2:k1, p: Eq k1 t1 t2]
       Leibniz k1 t1 ([x:k1] Eq k2 (f x) (f x))
       (refl_equal k2 (f t1)) t2 p;

type eqfNatS : (n1: Nat, n2:Nat) Eq Nat n1 n2 -> Eq Nat (S n1) (S n2) =
       [n1:Nat, n2:Nat]p: Eq Nat n1 n2]
       eqf Nat Nat S n1 n2 p;

Fix rightUnit [n:Nat] : Eq Nat n (plus n 0) =
       <: [n:Nat] Eq Nat n (plus n 0) :
       Cases n of
           0 => refl_equal Nat 0
       | S n' => eqfNatS n' (plus n' 0) (rightUnit n')
     End;

type eqTrans =
       [k: Prop][t1:k, t2: k, t3: k][p1: Eq k t1 t2][p2: Eq k t2 t3]
       Leibniz k t2 ([t:k] Eq k t1 t) p1 t3 p2;

Fix succPlus [n1:Nat]: (n2:Nat) Eq Nat (S (plus n1 n2)) (plus n1 (S n2)) =
       [n2:Nat]
       <: [n:Nat] Eq Nat (S (plus n n2)) (plus n (S n2)) :
       Cases n1 of
           0 => refl_equal Nat (S n2)
       | S n1' => eqfNatS (plus (S n1') n2)
           (plus n1' (S n2)) (succPlus n1' n2)
     End;

Fix plusSym [n1:Nat] : (n2:Nat) Eq Nat (plus n1 n2) (plus n2 n1) =
       [n2: Nat]
       <: [n:Nat] Eq Nat (plus n n2) (plus n2 n) :
       Cases n1 of
           0 => rightUnit n2
       | S n1' =>
           eqTrans Nat (plus (S n1') n2)
           (S (plus n2 n1')) (plus n2 (S n1'))
           eqfNatS (plus n1' n2) (plus n2 n1') (plusSym n1' n2)
           (succPlus n2 n1')
     End

Figure 3.9: More PL terms (Figure 3.8 continued)
3.3. SAMPLE VERO PROGRAMS

3.3.2 CL Programs

The following program defines a polymorphic list:

```verbatim
type rec list : Kind -> Kind =
  [a : Kind] nil | cons of a*(list a);
```

The sort Kind corresponds to the \( \Omega \) in the internal language. To simplify the syntax, we collapse the PL and the type language of CL, which can be distinguished by the sorts. The reserved word rec shows that the type list is a recursive type constructor.

We define the packaged nat type and the add function in the following program, as proposed in the TSCB paper [SSTP02].

```verbatim
type nat : Kind = <n : Nat> snat n;
val add : nat * nat -> nat =
  fn ns : nat * nat =>
    let val n1 = #1 ns
    val n2 = #2 ns
    open < n1', sn1 > = n1
    open < n2', sn2 > = n2
    in
    pack n = plus n1' n2'
    with sn1 + sn2 : snat n
  end
end;
```

The type expression \(<n : Nat> snat n\) represents the existential type \( \exists n : Nat. s\text{Nat}[n] \); expression open \(< n1', sn1 > = n1\) corresponds to the term let \((n'_1, sn_1) = n_1\) in ...; and the pack expression in the form of pack \(t = t1 with e : t2\) end corresponds to the term \((t = t_1, e : t_2)\).

Now we implement the function that computes the length of a function:

```verbatim
val a_zero : nat =
  pack n = 0
  with 0 : snat n
End;
val a_one : nat =
  pack n = S 0
  with 1 : snat n
End;
val rec length : (a:Kind)list a -> nat =
  [a:Kind]fn l : list a =>
    case l of nil => a_zero
    | cons x =>
      let
        val lth = length[a] (#2 x)
      in
        add (a_one, lth)
      End;
```
At present we do not support general pattern matching, which will be our future work.

What follows is the list type constructor indexed by its length, as shown in Section 2.3.2:

```ml
type rec llist : Kind -> Nat -> Kind =
[t:Kind, n:Nat]
  { lnil of <p1: Eq Nat 0 n > unit
    | lcons of < n' : Nat >
      < p2: Eq Nat (S n') n > t * (llist t n')
};
```

The curly brackets enclose the labeled sum type, which has similar syntax with the SML datatype declaration.

The problem of above list type constructor is that llist Nat (plus n1 n2) is treated as different type with llist Nat (plus n2 n1), because the PL terms plus n1 n2 and plus n2 n1 are two different normal forms and not βη-equivalent.

Here we implement the solution proposed in [SSTP02]. First we present a new version of llist:

```ml
type listP : Kind -> Nat -> Kind =
[t:Kind, n:Nat] < n' : Nat, p: Eq Nat n n' > llist t n';
```

The function listCvt converts a list of type llist a n to a list of type listP a n, and function cvt2 converts a list of type listP a (plus n1 n2) to a list of type listP a (plus n2 n1):

```ml
val listCvt : (t:Kind, n:Nat) llist t n -> listP t n =
[t:Kind, n:Nat]
  fn l : llist t n =>
    pack n' = n
    with (pack p = refl_equal Nat n
      with l : llist t n
        end) : < p: Eq Nat n n'> llist t n'
    end;
```

```ml
val cvt2 : (t:Kind, n1:Nat, n2:Nat) listP t (plus n1 n2) -> listP t (plus n2 n1) =
[t:Kind, n1:Nat, n2:Nat]
  fn l : listP t (plus n1 n2) =>
    let open < n1', pk > = l
      open < p, l' > = pk
      type pl1 = eqTrans Nat (plus n1 n1) (plus n1 n2)
        n1' (plusSym n2 n1) p
    in
      pack n' = n1'
      with (pack p = pl1
        with l' : llist t n1'
          End) : < p: Eq Nat (plus n2 n1) n'> llist t n'
    end
```

end
Chapter 4

Implementation

As part of the project, I implemented a type checker of Vero. For such a richly typed language like Vero, it is not an easy task to implement the type checker efficiently. As presented by Shao, League and Mommier [SLM98], even for such simple types like: \( T_n = T_{n_1} \to T_{n_2}, \ldots, T_1 = \alpha \to \alpha \), a naive representation of \( T_n \) like a tree structure would require \( O(2^n) \) space, so a sufficiently small \( n \) (e.g., 30) would wreak the efficiency of the compiler.

In this chapter, I give a detailed introduction of my implementation of the type checker. Most of the implementation is based on the work by Shao, League and Mommier [SLM98], and Saha [Sah02].

4.1 Implementing a PTS-like type system

From the syntax of the surface language of Vero, user may have noticed that we collapsed the PL and the type language of CL into one. It is mainly for the ease of syntax design. In our implementation, we stick with such strategy so that we can have a quick and clear implementation.

After the collapse, the whole type system is like a PTS with extension of inductive types, case analysis and fixpoints, recursive types, record and union types, and primitive types. The datatype \texttt{stem} used for representing type terms is shown below (we use the name \texttt{stem} because we are using suspension-based lambda encoding [Nad94] to represent types):

\[
\begin{array}{l}
\texttt{datatype} \quad \texttt{sort} \\
\quad \texttt{Kind} \\
\quad | \quad \texttt{Top} \\
\quad | \quad \texttt{Prop} \\
\quad | \quad \texttt{Type} \\
\quad | \quad \texttt{Ext}
\end{array}
\]

49
CHAPTER 4. IMPLEMENTATION

datatype stemfl
  - Var of int * stem (* variable & its kind *)
  - Sort of sort (* sorts *)
  - Tcon of tycons * stem (* primitive types and their kinds *)
  - All of stem * stem (* Pi type *)
  - Fn of stem * stem (* abstraction *)
  - App of stem * stem (* application *)
  - Seq of stem list (* auxiliary type, no counterpart in Vero *)
  - Ind of stem * int * stem list (* inductive definition *)
  - Con of stem * int * int * stem list (* constructors of inductive definition *)
  - Fix of int list * stem * int * stem list (* fixpoint *)
  - Case of stem * stem * stem list (* case analysis *)
  - Ex of stem * stem (* existential *)
  - Record of (symbol * stem) list (* record *)
  - Union of (symbol * stem) list (* union *)
  - Rec of stem * int (* μ *)
  - Env of stem * int * int * suspEnv (* suspension term *)
  - ITK of stem * stemfl (* for memoise *)
and
  - Dummy of int
  - Pair of stem * int
with
  - suspEnv — stem list
and
  - stem — stemfl hash_cell

... 

The datatype is very close to Saha’s implementation of PTS [Sah02], with some extension and revision. Here the sort Kind denotes $\Omega$ in CL, and Kind : Top. Prop, Type and Ext correspond to Prop, Type and Ext in PL, respectively.

We use deBruijn indices [De 80] for variables. Thus a term of the form

$$\lambda X : A.A \lambda Y : X \ldots Y \ldots X \ldots$$

is represented as

$$\lambda_\ldots : A.\lambda_\ldots : \#1 \ldots \#1 \ldots \#2 \ldots$$

where the variable is denoted by its lexical depth. This is useful because two $\alpha$-equivalent terms now have the same representation. Our representation of variables memoize more information than Saha’s implementation [Sah02]. Here instead of just memoizing the sort of variables, we memoize their kinds. Therefore when we see variable #2 in above type term, we do not have to go back to the first $\lambda$ to see it has kind $A$. Also since we use dag to represent type terms, this information redundancy will not consume much space (we need just one more reference).

Similarly, for primitive types, denoted as $Tcon(c, k)$, we also memoize the kind $k$ of the primitive type $c$.

Using deBruijn indices, the abstraction $\lambda X : A.B$ and production $\Pi X : A.B$ are denoted as $Fn(st_A, st_B)$ and $All(st_A, st_B)$ respectively, where $st_A$ and $st_B$
are representations of \(A\) and \(B\).

The mechanism for inductive definition is more general than that presented in Chapter 2. The implementation allows mutually recursive, parameterized inductive definitions. Also, in our implementation, we use \texttt{Case} and \texttt{Fix} to simulate the primitive recursion operator \texttt{Elim} presented in Chapter 2. The formation rules and the reduction rules for these constructs is entirely standard. This approach was formalized and proved equivalent to the primitive recursive approach in [Gim94]. We use the same formalization in our implementation. The implementation of \texttt{Ind}, \texttt{Con}, \texttt{Fix} and \texttt{Case} is the same with Saha’s [Sah02].

The existential type, record type, union type, and recursive type in CL are encoded using constructors \texttt{Ex}, \texttt{Record}, \texttt{Union} and \texttt{Rec}. The first three are standard treatment. The formalization of recursive type is similar to \texttt{Ind} and we also support mutual recursively defined types. The only difference between \texttt{Rec} and \texttt{Ind} is that we do not record the arguments in \texttt{Rec}, as we do in \texttt{Ind}.

The \texttt{Env} and \texttt{ITK} representation will be explained in the next two sections.

Here all the type terms are represented as dags. The datatype \texttt{sterm} is a reference pointing to a term of type \texttt{sterm1}, packaged with memoized information. It will be explained in following sections.

## 4.2 Suspension-based lambda encoding

Type reduction is done lazily in our implementation. To achieve this, we use the suspension-based lambda encoding, which was proposed in [Nad94] and used in the implementation of FLINT [SLM98] and Saha’s implementation of PTS system [Sah02]. Here we borrow Saha’s formulation, with some extension for CL terms.

The suspension terms are introduced by the \texttt{Env} constructor. Intuitively, these suspension terms are a form of closures: they are a pair of a type and an environment containing the values of the free variables. Reductions change the nesting depth of type terms which changes the deBruijn indices of variables. Therefore, the environment formed during a reduction must remember both the required substitutions and the change in DeBruijn indices. Accordingly, the environments are represented as a triple \((i, j, env)\) where the first index \(i\) indicates the current embedding level of the term, the second index \(j\) indicates the embedding level after the reduction, and the \(env\) is a mapping between variables (deBruijn indices) to terms.

Figure 4.2 shows how we use these suspension terms. Since the sort of a variable plays no role in the reductions, we represent variables only by their deBruijn index. During a \(\beta\)-reduction, instead of performing the substitution directly, we create a suspension term. The environment represents the following: the term \(p_2\) that was originally in the scope of 1 abstraction is now under none; \(p_3\) originally under the scope of 0 abstractions is to be substituted for the first free variable in \(p_2\). In the other cases we propagate the suspensions through to the subterms. This allows us to do the substitution lazily; we query the environment only when we encounter a variable during type manipulations. In
the case of a free variable, we simply adjust its DeBruijn index; in the case of a bound variable, we use the value from the environment.

4.3 Hash-consing and memoization

In our implementation, we hash-cons all types into one hash table. Under hash-consing, all types are guaranteed to use the most compact dag representation. Because we are using deBruijn notation, type variables are represented as integers and all $\alpha$-convertible types have identical representations, which allow them to be collapsed via hash-consing.

Each hash cell is defined as a mutable 4-tuple, with the following data type:

```
  type 'a hash_cell = (int * 'a * aux_info * 'a option ref) ref.
```

Remember the type `stem` is defined as `stem1 hash_cell`. Therefore a hash cell containing a type has the following fields: the first field is an integer hash code; the second is the actual denotation of the type; the third is a set of auxiliary information, which includes a flag that shows whether the type in the second field is in normal form or not, and if so, the set of free variables denoted in deBruijn indices. Compared with the implementation by [SLM98] and [Sah02], here we have one more field, i.e. an optional field of the kind of the type. After type checking a term, we memoize its kind so that we can avoid duplicate checking of $\alpha$-equivalent term.

We also memoize the result of every sequence of reduction of each term, as in [SLM98] and [Sah02]. Given a term `ref(h, t, aux, k)`, suppose $t$ can be reduced to $t'$; then, we can do in-place update, changing the second field of $t$ to a term $ITK(inj(t'), t)$, where $inj(t')$ is the hash cell containing the term $t'$.

4.4 Other issues to worked on

Although we have not run our type checker over benchmarks, according to the experience of [SLM98] and [Sah02], the combination of the lazy reduction, hash-consing and memoization is very effective: we have fast type equality operation, compact space usage, and linear-time traversal of types.

However, there still some issues for future work. One problem is to effectively represent `Nat` in PL. Since each natural number $\pi$ in CL has type $s_{\text{Nat}}[\tilde{n}]$. For a large natural number literal, we would have very inefficient denotation of its type. It is possible to introduce some compact notation of $\tilde{n}$ rather than the `unary` notation. If we need to destruct $\tilde{n}$, we can lazily unroll it back to the canonical denotation. Another problem is to efficiently implement the bound integer numbers and other data types in PL. This is essential to make Vero a practical language.
4.4. OTHER ISSUES TO WORKED ON

\[
\begin{align*}
\text{App}(Fn(p_1, p_2), p_3) & \leadsto Env(p_2, 1, 0, [\text{Pair}(p_3, 0)]) \quad (\beta - \text{redex}) \\
\text{Env}(\text{Var}(n, \_, i, j, \text{env})) & \leadsto \text{Var}(n - i + j) \quad n > i \quad (\text{free variable}) \\
\text{Env}(\text{Var}(n, \_, i, j, \text{env})) & \leadsto \text{Env}(p_1, 0, j - j', []) \quad n^{th} \text{element} \\
& \quad \text{of env} = \text{Pair}(j', p_1) \quad (\text{bound variable}) \\
\text{Env}(T\text{con} c, i, j, \text{env}) & \leadsto T\text{con} c \\
\text{Env}(\text{Sort} s, i, j, \text{env}) & \leadsto \text{Sort} s \\
\text{Env}(\text{Fn}(p_1, p_2), i, j, \text{env}) & \leadsto \text{Fn}(pr_{i,j}^\text{env}(p_1), ext_{i,j}^\text{env}(p_2)) \\
\text{Env}(\text{All}(p_1, p_2), i, j, \text{env}) & \leadsto \text{All}(pr_{i,j}^\text{env}(p_1), ext_{i,j}^\text{env}(p_2)) \\
\text{Env}(\text{App}(p_1, p_2), i, j, \text{env}) & \leadsto \text{App}(pr_{i,j}^\text{env}(p_1), pr_{i,j}^\text{env}(p_2)) \\
\text{Env}(\text{Ind}(p_1, i', p'), i, j, \text{env}) & \leadsto \text{Ind}(pr_{i,j}^\text{env}(p_1), i', pr_{i,j}^\text{env}(p')) \\
\text{Env}(\text{Con}(p_1, i', j', p'), i, j, \text{env}) & \leadsto \text{Con}(pr_{i,j}^\text{env}(p_1), i', j', pr_{i,j}^\text{env}(p')) \\
\text{Env}(\text{Fix}(\bar{i}, p_1, i', p'), i, j, \text{env}) & \leadsto \text{Fix}(\bar{i}, pr_{i,j}^\text{env}(p_1), i', pr_{i,j}^\text{env}(p')) \\
\text{Env}(\text{Case}(p_1, p_2, p'), i, j, \text{env}) & \leadsto \text{Case}(pr_{i,j}^\text{env}(p_1), pr_{i,j}^\text{env}(p_2), pr_{i,j}^\text{env}(p')) \\
\text{Env}(\text{Seq}(p'), i, j, \text{env}) & \leadsto \text{Seq}(pr_{i,j}^\text{env}(p')) \\
\text{Env}(\text{Ex}(p_1, p_2), i, j, \text{env}) & \leadsto \text{Ex}(pr_{i,j}^\text{env}(p_1), pr_{i,j}^\text{env}(p_2)) \\
\text{Env}(\text{Record}(l_i, p_1), i, j, \text{env}) & \leadsto \text{Record}(l_i, pr_{i,j}^\text{env}(p_1)) \\
\text{Env}(\text{Union}(l_i, p_1), i, j, \text{env}) & \leadsto \text{Union}(l_i, pr_{i,j}^\text{env}(p_1)) \\
\text{Env}(\text{Rec}(p, i'), i, j, \text{env}) & \leadsto \text{Rec}(pr_{i,j}^\text{env}(p), i')
\end{align*}
\]

where

\[
\begin{align*}
pr_{i,j}^\text{env}(p) & \equiv \text{Env}(p, i, j, \text{env}) \\
ext_{i,j}^\text{env}(p) & \equiv \text{Env}(p, i + 1, j + 1, (\text{Dummy} \; j :: \text{env}))
\end{align*}
\]

Figure 4.1: Using suspension terms for lazy reduction
Chapter 5

Conclusion and Future Work

In this report we presented the design and implementation of the Vero language. Both the internal (core) language and the surface (core) language of Vero are presented in the report. We also formulated the static and dynamic semantics, and proved the type safety of Vero. Some sample programs were presented to show the expressiveness of the language.

The current version of Vero is still a quite simplified version, in which we just concentrate on the design of the strong type system of the core language, and explore the expressiveness of such a language. A lot of orthogonal but important issues are omitted. To make Vero a practical language, we need to solve the following problems in our future work:

1. At present we use CIC as the meta-logical framework. Although CIC provides a powerful meta-logic, it has a lot of restrictions for representation, such as the requirement of positiveness in inductive definitions. This makes it sometimes clumsy to use. In the future we will explore the possibility to have a MLF with the expressive power of the logical framework (e.g., LF) and the reasoning power of a modern proof assistant (e.g., Coq). Such a MLF will also pose new challenges to our design of the computation language. For instance, if an MLF representation $d$ uses higher-order abstract syntax, how do we use its terms to index a datatype definition $D$ (which usually uses first-order representations)?

2. Vero is the first language that supports explicit handling of proof objects. A Vero program may contain both MLF terms (e.g., inductive definitions, primitive recursive functions) and usual Vero computation terms (e.g., concrete datatype definitions, recursive functions). We want to extend and adapt the ML-style module system to provide scalable support to large-scale CPL programs. Actually, since we have existential types in CL, which can package either types in CL or MLF terms, we have the potential to support first-class module systems in Vero. However, that would destroy the current clean design, which does not have circular dependency. That is, in our current design, the computation term might depend on
types or MLF terms, while a type only depends on MLF terms, and MLF terms have no dependency with computation terms and types. If we add first-class module system, types and MLF terms will in turn depend on modules, which are now first-class computation terms. We need some research to introduce some clean formulation and make sure this will not introduce inconsistency.

3. Indexed data type definitions such as singleton integer types and guarded recursive data types [XCC03] are the key facility to link the computation term with the MLF representation. We want to explore its design space and see how different decisions affect the usability and expressiveness. We’ll also design more powerful pattern matching support to make programming with these data types easier.

4. Both the PL and the CL of the current version of Vero is explicitly typed. To make it more easy to use, we need to provide partial type inference support, as did in Coq [Tea03]. Also the way to define indexed data types requires repeated work (see our definition of singleton list in Section 2.3.2). Given a PL inductive definition, it would be desirable to automatically generate the corresponding indexed datatype definition. Although we have proposed some approaches in this report, they are just experimental approaches and need further research.

5. There are some other implementation issues to be worked on, such as add exception handling and reference to the computation language, build libraries for both PL and CL, or more preferably, develop tools that can transform SML and Coq code to Vero CL and PL code so that we can take advantage of existing SML and Coq libraries.
Bibliography


Appendix A

Type Safety of CL

Lemma A.1 (Canonical Forms) Suppose that \( v \) is a well-formed value and we have \( \vdash \vdash \vdash v : \tau \).

1. If \( \vdash \vdash \vdash v : \text{unit} \), then \( v \equiv () \).
2. If \( \vdash \vdash \vdash v : s_{\text{Nat}}[A] \), then \( v \equiv \pi\overline{n} \) and \( \overline{n} = _{\beta n} A \), for some \( n \).
3. If \( \vdash \vdash \vdash v : s_{\text{Bool}}[A] \), then either \( v \equiv \text{tt} \) and \( A = _{\beta n} \text{true} \), or \( v \equiv \text{ff} \) and \( A = _{\beta n} \text{false} \).
4. If \( \vdash \vdash \vdash v : \tau_1 \rightarrow \tau_2 \), then \( v \equiv \lambda x : \tau_1.e \) for some \( x \) and \( e \).
5. If \( \vdash \vdash \vdash v : \forall t : \kappa.\tau \), then \( v \equiv \Lambda t : \kappa.e \) for some \( e \).
6. If \( \vdash \vdash \vdash v : \forall X : A.\tau \), then \( v \equiv \Lambda X : A.e \) for some \( e \).
7. If \( \vdash \vdash \vdash v : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \), then \( v \equiv \{l_1 = v_1, \ldots, l_n = v_n\} \) for some \( v_1, \ldots, v_n \).
8. If \( \vdash \vdash \vdash v : \tau_1 + \tau_2 \), then either \( v \equiv \text{inj}_{1}^{1+\tau_2}v_1 \) for some \( v_1 \), or \( v \equiv \text{inj}_{2}^{1+\tau_2}v_2 \) for some \( v_2 \).
9. If \( \vdash \vdash \vdash v : \mu_\kappa(\tau_1, \tau_2) \), then \( v \equiv \text{fold}_{1}^{\mu_\kappa(\tau_1, \tau_2)}v' \) for some \( v' \).
10. If \( \vdash \vdash \vdash v : \exists t : \tau.\tau' \), then \( v \equiv \langle t = \tau, v' : \tau' \rangle \) for some \( v' \).
11. If \( \vdash \vdash \vdash v : \exists X : A.\tau \), then \( v \equiv \langle X = A, v' : \tau \rangle \) for some \( v' \).

Proof. We can draw the conclusion by a simple inspection of the expression formation rules shown in Figure 2.8 and 2.9, using the fact that \( v \) is a value. \( \square \)

Theorem A.2 (Progress) If \( \vdash \vdash \vdash e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e \rightarrow e' \).

Proof. We prove it by induction on the type derivation for \( e \). We analyze the cases corresponding to the last expression formation rule used in the type derivation.
(Rule E-VAR) Cannot occur, since we require empty contexts.

(Rules E-UNIT, E-NAT, E-TRUE, E-FALSE, E-FN, E-TFN1 and E-TFN2) In each case $e$ is a value, which completes the proof.

(Rule E-ADD) We require that $e \equiv e_1 + e_2$ for some $e_1$ and $e_2$, and $\vdash \cdots \vdash e_1 : s_{\text{Nat}}[A_1]$ and $\vdash \cdots \vdash e_2 : s_{\text{Nat}}[A_2]$, and $\tau \equiv s_{\text{Nat}}[\text{plus } A_1 A_2]$. By induction hypothesis, there are three cases.

- $e_1$ and $e_2$ are values. According to Lemma A.1, there exist $n_1$ and $n_2$ such that $e_1 \equiv \frac{n_1}{\tau_1}$ and $e_2 \equiv \frac{n_2}{\tau_2}$. By reduction rule R-ADD, $e_1 + e_2 \hookrightarrow n_1 + n_2$. Therefore $e \mapsto \bullet\{n_1 + n_2\}$.

- $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1\{e_{11}\}$ and $e'_1 \equiv E_1\{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \hookrightarrow e'_{11}$. Then $e \mapsto E\{e'_{11}\}$, where $E \equiv E_1 + e_2$.

- $e_1$ is a value and $e_2 \mapsto e'_2$ for some $e'_2$. The proof is similar to the above case.

(Rule E-LT) We require that $e \equiv e_1 \lt e_2$ for some $e_1$ and $e_2$, and $\vdash \cdots \vdash e_1 : s_{\text{Nat}}[A_1]$ and $\vdash \cdots \vdash e_2 : s_{\text{Nat}}[A_2]$, and $\tau \equiv s_{\text{Bool}}[\text{lt } A_1 A_2]$.

The following proof is similar to the above case.

(Rule E-IF) We require that $e \equiv \text{if } [B, A](e_0, X_1, e_1, X_2, e_2)$, and $\vdash \cdots \vdash e_0 : s_{\text{Bool}}[A']$. By induction hypothesis, there are two cases.

- $e_0$ is a value. According to Lemma A.1, either $e_0 \equiv \text{tt}$ or $e_0 \equiv \text{ff}$. If $e_0 \equiv \text{tt}$, by reduction rule R-IF-T, if $[B, A](e_0, X_1, e_1, X_2, e_2) \hookrightarrow [A/X_1]e_1$. Therefore $e \mapsto \bullet\{[A/X_1]e_1\}$. Otherwise, by reduction rule R-IF-F we know if $[B, A](e_0, X_1, e_1, X_2, e_2) \hookrightarrow [A/X_2]e_2$. Therefore $e \mapsto \bullet\{[A/X_2]e_2\}$.

- $e_0 \mapsto e'_0$ for some $e'_0$. Therefore $e_0 \equiv E_0\{e_{01}\}$ and $e'_0 \equiv E_0\{e'_{01}\}$ for some evaluation context $E_0$ and redex $e_{01}$ such that $e_{01} \hookrightarrow e'_{01}$. Then $e \mapsto E\{e'_{01}\}$, where $E \equiv \text{if } [B, A](E_0, X_1, e_1, X_2, e_2)$.

(Rule E-FIX) We require that $e \equiv \text{fix } x : \tau.f$ for some $x$ and $f$. According to the reduction rule R-FIX, $\text{fix } x : \tau.f \hookrightarrow [\text{fix } x : \tau.f/x]f$. Therefore $e \mapsto \bullet\{[\text{fix } x : \tau.f/x]f\}$.

(Rule E-APP) We require that $e \equiv e_1 e_2$ for some $e_1$ and $e_2$. By induction hypothesis, there are three cases.

- $e_1$ and $e_2$ are values. According to Lemma A.1, there exist $\tau_1$, $x$ and $e'_1$ such that $e_1 \equiv \lambda x : \tau_1.e'_1$. By reduction rule R-\(\beta\), $e_1 e_2 \hookrightarrow [e_2/x]e'_1$. Therefore $e \mapsto \bullet\{[e_2/x]e'_1\}$.
• $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1 \{e_{11}\}$ and $e'_1 \equiv E_1 \{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \mapsto e'_{11}$. Then $e \mapsto E \{e'_{11}\}$, where $E \equiv E_1 e_2$.

• $e_1$ is a value and $e_2 \mapsto e'_2$ for some $e'_2$. The proof is similar to the above case.

(Rule E-tapp1) We require that $e \equiv e_0[\tau_0]$ and $\tau \equiv [\tau_0/t]\tau'$ for some $e_0$, $\tau_0$, $t$ and $\tau'$. By induction hypothesis, there are two cases.

• $e_0$ is a value. According to Lemma A.1, there exist $\kappa$ and $e_1$ such that $e_0 \equiv \Lambda : \kappa \cdot e_1$. By reduction rule R-tapp1, $e_0[\tau_0] \mapsto [\tau_0/t]e_1$. Therefore $e \mapsto \bullet ([\tau_0/t]e_1)$.

• $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1 \{e_{11}\}$ and $e'_1 \equiv E_1 \{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \mapsto e'_{11}$. Then $e \mapsto E \{e'_{11}\}$, where $E \equiv E_1[\tau_0]$.

(Rule E-tapp2) Similar to the proof of the above case.

(Rule E-str) We require that $e \equiv \{l_1 = e_1, \ldots, l_n = e_n\}$ and $\tau \equiv \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ for some $l_i, e_i$ and $\tau_i$, ($1 \leq i \leq n$). By induction hypothesis, there are two cases.

• For $i = 1, \ldots, n$, all the $e_i$ are values. Then $e$ is itself a value.

• Without loss of generality, we assume that the first expression among all the $e_i$ ($i = 1, \ldots, n$) that is not a value is $e_j$, ($1 \leq j \leq n$). $e_j \mapsto e'_j$ for some $e'_j$. Therefore $e_j \equiv E_1 \{e_{j}, \ldots, e_{n}\}$ and $e'_j \equiv E_1 \{e'_{j}, \ldots, e_{n}\}$ for some evaluation context $E_1$ and redex $e_{j}$ such that $e_{j} \mapsto e'_{j}$. Then $e \mapsto E \{e'_{j}\}$, where $E \equiv \{l_1 = e_1, \ldots, l_{j-1} = e_{j-1}, l_j = E_1, l_{j+1} = e_{j+1}, \ldots, l_n = e_n\}$ (Note that $e_1, \ldots, e_{j-1}$ are all values).

(Rule E-proj) We require that $e \equiv e_0.l_j$ for some $e_0$ and $l_j$. By induction hypothesis, there are two cases.

• $e_0$ is a value. According to Lemma A.1, there exist $l_i$ and $v_i$ for $i = 1, \ldots, n$, such that $e_0 \equiv \{l_1 = v_1, \ldots, l_n = v_n\}$, and $1 \leq j \leq n$. By reduction rule R-proj, $e_0.l_j \mapsto v_j$. Therefore $e \mapsto \bullet (v_j)$.

• $e_0 \mapsto e'_0$ for some $e'_0$. Therefore $e_0 \equiv E_1 \{e_{0j}\}$ and $e'_0 \equiv E_1 \{e'_{0j}\}$ for some evaluation context $E_1$ and redex $e_{0j}$ such that $e_{0j} \mapsto e'_{0j}$. Then $e \mapsto E \{e'_{0j}\}$, where $E \equiv E_1.l_j$. 

63
(Rule E-INJL) We require that $e \equiv \text{inj}_{\tau_1 \rightarrow \tau_2}^{\tau_1} e_1$ for some $\tau_1$, $\tau_2$ and $e_1$. By induction hypothesis, there are two cases.

- $e_1$ is a value. Then $e$ is itself a value.
- $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1 \{e_{11}\}$ and $e'_1 \equiv E_1 \{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \mapsto e'_{11}$. Then $e \mapsto E \{e'_{11}\}$, where $E \equiv \text{inj}_{\tau_1 \rightarrow \tau_2}^{\tau_1} E_1$.

(Rule E-INJR) The proof is similar to the above case.

(Rule E-CASE) We require that $e \equiv \text{case} e_0, e_1, e_2$ for some $e_0$, $e_1$ and $e_2$. By induction hypothesis, there are two cases.

- $e_0$ is a value. According to Lemma A.1, either $e_0 \equiv \text{inj}_{\tau_1 \rightarrow \tau_2}^{\tau_1} v_1$ or $e_0 \equiv \text{inj}_{\tau_1 \rightarrow \tau_2}^{\tau_2} v_2$ for some $v_1$, $v_2$, $\tau_1$ and $\tau_2$. Accordingly, by reduction rule R-CASE1 and R-CASE2, we have either $e \mapsto e_1 v_1$ or $e \mapsto e_2 v_2$. Therefore either $e \mapsto \bullet \{e_1 v_1\}$ or $e \mapsto \bullet \{e_2 v_2\}$.
- $e_0 \mapsto e'_0$ for some $e'_0$. Therefore $e_0 \equiv E_1 \{e_{00}\}$ and $e'_0 \equiv E_1 \{e'_{00}\}$ for some evaluation context $E_1$ and redex $e_{00}$ such that $e_{00} \mapsto e'_{00}$. Then $e \mapsto E \{e'_{00}\}$, where $E \equiv \text{case} E_1, e_1, e_2$.

(Rule E-FOLD) We require that $e \equiv \text{fold}^{\mu_{\kappa}(\tau_1, \tau_2)} \kappa \tau_1 \tau_2 e_1$ for some $\kappa$, $\tau_1$, $\tau_2$ and $e_1$. By induction hypothesis, there are two cases.

- $e_1$ is a value. Then $e$ is itself a value.
- $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1 \{e_{11}\}$ and $e'_1 \equiv E_1 \{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \mapsto e'_{11}$. Then $e \mapsto E \{e'_{11}\}$, where $E \equiv \text{fold}^{\mu_{\kappa}(\tau_1, \tau_2)} \kappa \tau_1 \tau_2 E_1$.

(Rule E-UNFLD) We require that $e \equiv \text{unfold} e_1$ and $\tau \equiv \mathsf{K}(\kappa, \tau_1, \tau_2)$ for some $\kappa$, $\tau_1$, $\tau_2$ and $e_1$. By induction hypothesis, there are two cases.

- $e_1$ is a value. According to Lemma A.1, $e_1 \equiv \text{fold}^{\mu_{\kappa}(\tau_1, \tau_2)} \kappa \tau_1 \tau_2 v_1$. By reduction rule R-UNFD, $e \mapsto v_1$. Therefore $e \mapsto \bullet \{v_1\}$.
- $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1 \{e_{11}\}$ and $e'_1 \equiv E_1 \{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \mapsto e'_{11}$. Then $e \mapsto E \{e'_{11}\}$, where $E \equiv \text{unfold} E_1$.

(Rule E-PACK1) We require that $e \equiv \langle t = \tau, e_1 : \tau' \rangle$ and $\tau \equiv \exists t : \kappa. \tau'$. By induction hypothesis, there are two cases.

- $e_1$ is a value. Then $e$ is itself a value.
- $e_1 \mapsto e'_1$ for some $e'_1$. Therefore $e_1 \equiv E_1 \{e_{11}\}$ and $e'_1 \equiv E_1 \{e'_{11}\}$ for some evaluation context $E_1$ and redex $e_{11}$ such that $e_{11} \mapsto e'_{11}$. Then $e \mapsto E \{e'_{11}\}$, where $E \equiv \langle t = \tau, E_1 : \tau' \rangle$. 
(Rule E-PACK2) The proof is similar to the above case.

(Rule E-LET) We require that \( e \equiv \text{let } x = e_1 \text{ in } e_2 \) for some \( x, e_1 \) and \( e_2 \). By induction hypothesis, there are two cases.

- \( e_1 \) is a value. By reduction rule R-LET, \( e \mapsto [e_1/x]e_2 \). Therefore \( e \mapsto \bullet[[e_1/x]e_2] \).
- \( e_1 \mapsto e'_1 \) for some \( e'_1 \). Therefore \( e_1 \equiv E_1\{e_{11}\} \) and \( e'_1 \equiv E_1\{e'_{11}\} \) for some evaluation context \( E_1 \) and redex \( e_{11} \) such that \( e_{11} \mapsto e'_{11} \). Then \( e \mapsto E\{e'_{11}\} \), where \( E \equiv \text{let } x = E_1 \text{ in } e_2 \). 

(Rule E-TLET1) We require that \( e \equiv t = \tau' \) in \( e_1 \) for some \( t, \tau' \) and \( e_1 \). By reduction rule R-TLET1, \( e \mapsto [\tau'/t]e_1 \). Therefore \( e \mapsto \bullet[[\tau'/t]e_1] \).

(Rule E-TLET2) The proof is similar to the above case.

(Rule E-OPEN1) We require that \( e \equiv \text{let}(t, x) = e_1 \) in \( e_2 \) for some \( t, x, e_1 \) and \( e_2 \). By induction hypothesis, there are two cases.

- \( e_1 \) is a value. According to Lemma A.1, \( e_1 \equiv t' = \tau_1, v : \tau_2 \). By reduction rule R-OPEN1, \( e \mapsto [v/x][\tau_1/t]e_2 \). Therefore \( e \mapsto \bullet[[v/x][\tau_1/t]e_2] \).
- \( e_1 \mapsto e'_1 \) for some \( e'_1 \). Therefore \( e_1 \equiv E_1\{e_{11}\} \) and \( e'_1 \equiv E_1\{e'_{11}\} \) for some evaluation context \( E_1 \) and redex \( e_{11} \) such that \( e_{11} \mapsto e'_{11} \). Then \( e \mapsto E\{e'_{11}\} \), where \( E \equiv \text{let}(t, x) = E_1 \text{ in } e_2 \).

(Rule E-OPEN2) The proof is similar to the above case.

(Rule E-CONV) By induction hypothesis. 

\[ \square \]

Lemma A.3 (Type Inversion)

1. If \( \Gamma; \Delta; \Sigma \vdash x : \tau \), then \( \tau = \tau, \Sigma(x) \) and \( \Gamma; \Delta \vdash \Sigma \text{ ok} \).
2. If \( \Gamma; \Delta; \Sigma \vdash () : \tau \), then \( \tau = \text{unit} \) and \( \Gamma; \Delta \vdash \Sigma \text{ ok} \).
3. If \( \Gamma; \Delta; \Sigma \vdash \pi : \tau \), then \( \tau = \pi, \text{Nat}[\widehat{\pi}] \) and \( \Gamma; \Delta \vdash \Sigma \text{ ok} \).
4. If \( \Gamma; \Delta; \Sigma \vdash \text{tt} : \tau \), then \( \tau = \text{true}, \text{Bool}[\text{true}] \) and \( \Gamma; \Delta \vdash \Sigma \text{ ok} \).
5. If \( \Gamma; \Delta; \Sigma \vdash \text{ff} : \tau \), then \( \tau = \text{false}, \text{Bool}[\text{false}] \) and \( \Gamma; \Delta \vdash \Sigma \text{ ok} \).
6. If \( \Gamma; \Delta; \Sigma \vdash e_1 \cdot e_2 : \tau \), then \( \tau = \text{Nat}([\text{plus} A_1 A_2], \Gamma; \Delta; \Sigma \vdash e_1 : \text{Nat}[A_1], \) and \( \Gamma; \Delta; \Sigma \vdash e_2 : \text{Nat}[A_2], \) for some \( A_1 \) and \( A_2 \).
7. If \( \Gamma; \Delta; \Sigma \vdash e_1 \cdot e_2 : \tau \), then \( \tau = \text{Nat}([\text{lt} A_1 A_2], \Gamma; \Delta; \Sigma \vdash e_1 : \text{Nat}[A_1], \) and \( \Gamma; \Delta; \Sigma \vdash e_2 : \text{Nat}[A_2], \) for some \( A_1 \) and \( A_2 \).
8. If $\Gamma; \Delta; \Sigma \vdash \text{if}[B, A](e_0, X_1.e_1, X_2.e_2) : \tau$, then $\Gamma \vdash B : \text{bool} \to \text{Prop}$,
$\Gamma; \Delta; \Sigma \vdash e_0 : \text{bool}(A')$, $\Gamma \vdash A : B \ A'$, $\Gamma, X_1 : B \ \text{true}; \Delta; \Sigma \vdash e_1 : \tau'$, $\Gamma, X_2 : B \ \text{false}; \Delta; \Sigma \vdash e_2 : \tau'$, $\Gamma, \Delta \vdash \tau' : \Omega$, and $\tau = t \tau'$ for some $A'$ and $\tau'$.

9. If $\Gamma; \Delta; \Sigma \vdash \lambda x.\tau_1.e : \tau$, then $\tau = t \tau_1 \tau_2$, and $\Gamma; \Delta; \Sigma, x : \tau_1 \vdash e : \tau_2$.

10. If $\Gamma; \Delta; \Sigma \vdash \text{let}\; \kappa.e : \tau$, then there exists $\tau'$ such that $\Gamma; \Delta, t : \kappa; \Sigma \vdash e : \tau'$, $\Gamma; \Delta \vdash \Sigma \text{ ok and } \tau = t \forall \kappa.\tau'$.

11. If $\Gamma; \Delta; \Sigma \vdash \text{let}\; \kappa.X.e : \tau$, then there exists $\tau'$ such that $\Gamma \vdash A : s$ $(s \neq \text{Ext})$, $\Gamma; \Delta \vdash \Sigma \text{ ok, } \Gamma, X : A ; \Delta; \Sigma \vdash e : \tau'$, and $\tau = t \forall X.X.\tau'$.

12. If $\Gamma; \Delta; \Sigma \vdash \text{fix}\; x.\tau'.f : \tau$, then $\tau = t \tau'$, $\Gamma; \Delta \vdash \tau : \Omega$, and $\Gamma; \Delta; \Sigma, x : \tau' \vdash f : \tau'$.

13. If $\Gamma; \Delta; \Sigma \vdash e_1.e_2 : \tau$, then there exist $\tau_1$ and $\tau_2$ such that $\Gamma; \Delta; \Sigma \vdash e_1 : \tau_1 \to \tau_2$, $\Gamma; \Delta; \Sigma \vdash e_2 : \tau_1$, and $\tau = t \tau_2$.

14. If $\Gamma; \Delta; \Sigma \vdash e[\tau_1] : \tau$, then there exist $\tau_2$ and $\kappa$ such that $\Gamma; \Delta; \Sigma \vdash e : \forall t.\kappa.\tau_2$, and $\tau = t \tau_2$.

15. If $\Gamma; \Delta; \Sigma \vdash e[A] : \tau$, then there exist $B, s$ $(s \neq \text{Ext})$ and $\tau'$ such that $\Gamma \vdash A : B$, $\Gamma \vdash B : s$, $\Gamma; \Delta; \Sigma \vdash e : \forall X.B.\tau'$ and $\tau = t [A/X] \tau'$.

16. If $\Gamma; \Delta; \Sigma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \tau$, then there exist $\tau_1, \ldots, \tau_n$ such that $\tau = t \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ and for $i = 1, \ldots, n$ $\Gamma; \Delta; \Sigma \vdash e_i : \tau_i$.

17. If $\Gamma; \Delta; \Sigma \vdash e.l_i : \tau$, then there exist $\tau_1, \ldots, \tau_n$ $(1 \leq i \leq n)$ such that $\Gamma; \Delta; \Sigma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ and $\tau = t \tau_i$.

18. If $\Gamma; \Delta; \Sigma \vdash \text{inj}_1^{\tau_1 + \tau_2}e : \tau$, then $\tau = t \tau_1 + \tau_2$, $\Gamma; \Delta \vdash \tau_1 + \tau_2 : \Omega$, and $\Gamma; \Delta; \Sigma \vdash e : \tau_1$.

19. If $\Gamma; \Delta; \Sigma \vdash \text{inj}_2^{\tau_1 + \tau_2}e : \tau$, then $\tau = t \tau_1 + \tau_2$, $\Gamma; \Delta \vdash \tau_1 + \tau_2 : \Omega$, and $\Gamma; \Delta; \Sigma \vdash e : \tau_2$.

20. If $\Gamma; \Delta; \Sigma \vdash \text{case}(e_0, e_1, e_2) : \tau$, then there exist $\tau'$, $\tau_1$ and $\tau_2$ such that $\Gamma; \Delta; \Sigma \vdash e_0 : \tau_1 + \tau_2$, $\Gamma; \Delta; \Sigma \vdash e_1 : \tau_1 \to \tau'$, $\Gamma; \Delta; \Sigma \vdash e_2 : \tau_2 \to \tau'$, and $\tau = t \tau'$.

21. If $\Gamma; \Delta; \Sigma \vdash \text{fold}^{\mu_\kappa}(\tau_1, \tau_2)e : \tau$, then $\Gamma; \Delta \vdash \mu_\kappa(\tau_1, \tau_2) : \Omega$, $\Gamma; \Delta; \Sigma \vdash e : \mathcal{K}(\kappa, \tau_1, \tau_2)$, and $\tau = t \mu_\kappa(\tau_1, \tau_2)$.

22. If $\Gamma; \Delta; \Sigma \vdash \text{unfold}e : \tau$, then there exist $\kappa, \tau_1$, and $\tau_2$ such that $\Gamma; \Delta; \Sigma \vdash e : \mu_\kappa(\tau_1, \tau_2)$, and $\tau = t \mathcal{K}(\kappa, \tau_1, \tau_2)$.

23. If $\Gamma; \Delta; \Sigma \vdash (t = \tau_1, e : \tau_2) : \tau$, then there exist $\kappa$ such that $\Gamma; \Delta \vdash \tau_1 : \kappa$, $\Gamma; \Delta; \Sigma \vdash e : [\tau_1/t] \tau_2$, $\Gamma; \Delta; t : \kappa \vdash \tau_2 : \Omega$, and $\tau = t \exists \kappa.\tau_2$. 
24. If $\Gamma; \Delta; \Sigma \vdash \langle X = A, e : \tau' \rangle : \tau$, then there exist $B$ and $s$ ($s \neq \text{Ext}$) such that $\Gamma \vdash A : B$, $\Gamma \vdash B : s$, $\Gamma, X : B; \Delta \vdash \tau' : \Omega$, $\Gamma; \Delta; \Sigma \vdash e : [A/X]\tau'$, and $\tau =_{t} \exists x : B. \tau'$.

25. If $\Gamma; \Delta; \Sigma \vdash x = e_0$ in $e : \tau$, then there exist $\tau_0$ and $\tau'$ such that $\Gamma; \Delta; \Sigma \vdash e_0 : \tau_0$, $\Gamma; \Delta; \Sigma, x : \tau_0 \vdash e : \tau'$, and $\tau =_{t} \tau'$.

26. If $\Gamma; \Delta; \Sigma \vdash t = \tau_0$ in $e : \tau$, then there exist $\kappa$ and $\tau'$ such that $\Gamma; \Delta \vdash \tau_0 : \kappa$, $\Gamma; \Delta; \Sigma \vdash [\tau_0/t]e : \tau'$, and $\tau =_{t} \tau'$.

27. If $\Gamma; \Delta; \Sigma \vdash X = A$ in $e : \tau$, then there exist $B$ and $\tau'$ such that $\Gamma \vdash A : B$, $\Gamma; \Delta; \Sigma \vdash [A/X]e : \tau'$, and $\tau =_{t} \tau'$.

28. If $\Gamma; \Delta; \Sigma \vdash (let(t, x) = e_0$ in $e : \tau$, then there exist $\kappa$, $\tau_0$ and $\tau'$ such that $\Gamma; \Delta \vdash \tau' : \Omega$, $\Gamma; \Delta; \Sigma \vdash e_0 : \exists t : \kappa. \tau_0$, $\Gamma; \Delta, t : \kappa; \Sigma, x : \tau_0 \vdash e : \tau'$, and $\tau =_{t} \tau'$.

29. If $\Gamma; \Delta; \Sigma \vdash \text{let}(X, x) = e_0$ in $e : \tau$, then there exist $A$, $\tau_0$ and $\tau'$ such that $\Gamma; \Delta \vdash \tau' : \Omega$, $\Gamma; \Delta; \Sigma \vdash e_0 : \exists X : A. \tau_0$, $\Gamma, X : A; \Delta, \Sigma, x : \tau_0 \vdash e : \tau'$, and $\tau =_{t} \tau'$.

Proof. By inspection of the expression formation rules, shown in Figure 2.8 and 2.9, we can see that every typing judgement $\Gamma; \Delta; \Sigma \vdash e : \tau$ is the result of applying a unique typing rule, apart from the type-equality rule $\text{E-conv}$. Due to the reflexivity and transitivity of $=_{t}$, any derivation of $\Gamma; \Delta; \Sigma \vdash e : \tau$ can be converted to a standard form in which there is an application of the rule $\text{E-conv}$ at its root, the derivation of whose first premise ends with an instance of a rule other than $\text{E-conv}$. This rule is uniquely determined by the typing judgement. \hfill\Box

Lemma A.4 (Term Substitution) If $\Gamma; \Delta; \Sigma, x : \tau' \vdash e : \tau$, and $\Gamma; \Delta; \Sigma \vdash e' : \tau'$, we must have $\Gamma; \Delta; \Sigma \vdash [e'/x]e : \tau$.

Proof. By induction on the type derivation of $\Gamma; \Delta; \Sigma, x : \tau', \Sigma_2 \vdash e : \tau$ (the Proposition 2.12 is used). \hfill\Box

Lemma A.5 (Type Substitution I) If $\Gamma; \Delta, t : \kappa; \Sigma \vdash e : \tau$, and $\Gamma; \Delta \vdash \tau' : \kappa$, we must have $\Gamma; \Delta; [\tau'/t] \Sigma \vdash [\tau'/t]e : [\tau'/t] \tau$.

Proof. By induction on the derivation of $\Gamma; \Delta, t : \kappa, \Delta_2; \Sigma \vdash e : \tau$ (Propositions 2.12, 2.8, and 2.6 are used). \hfill\Box

Lemma A.6 (Type Substitution II) If $\Gamma_1, X : B, \Gamma_2; \Delta; \Sigma \vdash e : \tau$, and $\Gamma_1 \vdash A : B$, we must have $\Gamma_1, [A/X] \Gamma_2; [A/X] \Delta; [A/X] \Sigma \vdash [A/X] e : [A/X] \tau$. 

Proof. By induction on the derivation of \( \Gamma_1, X : B, \Gamma_2; \Delta; \Sigma \vdash e : \tau \) (Propositions 2.2.2, 2.12, 2.9 and 2.6 are used).

Before we prove the subject reduction lemma, we need firstly prove the adequacy of the representation of arithmetic using our proof language.

**Lemma A.7 (Adequacy of the PL representation of arithmetic)**

1. For all \( n_1, n_2 \in \mathbb{N} \), plus \( \hat{n_1} \hat{n_2} = \beta_{\mu} n_1 + n_2 \)
2. For all \( n_1, n_2 \in \mathbb{N} \), lt \( \hat{n_1} \hat{n_2} = \beta_{\mu} \text{true} \) if and only if \( n_1 < n_2 \).
3. For all \( n_1, n_2 \in \mathbb{N} \), le \( \hat{n_1} \hat{n_2} = \beta_{\mu} \text{false} \) if and only if \( n_1 \geq n_2 \).

Proof sketch.

1. By induction on \( n_1 \) and inspection of the definition of plus.
2. By induction on \( n_1 \) and inspection of the definition of lt and le. We need first prove that For all \( n_1, n_2 \in \mathbb{N} \), le \( \hat{n_1} \hat{n_2} = \beta_{\mu} \text{true} \) if and only if \( n_1 \leq n_2 \).
3. By 2 and the definition of lt.

**Lemma A.8 (Subject Reduction)** If \( \vdash \vdash e : \tau \) and \( e \rightarrow e' \), then \( \vdash \vdash e' : \tau \).

Proof. By induction on the rules defining reduction relation \( \rightarrow \).

(Rule R-ADD) We require \( e \equiv \overline{n_1 + n_2} \) for some \( n_1 \) and \( n_2 \). By Lemma A.3, we have \( \tau =_{\iota} \text{snat}[\text{plus } A_1 A_2] \), \( \vdash \vdash \overline{n_1} : \text{snat}[A_1] \), and \( \vdash \vdash \overline{n_2} : \text{snat}[A_2] \), for some \( A_1 \) and \( A_2 \). Again by Lemma A.3, \( \text{snat}[A_1] =_{\iota} \text{snat}[\overline{n_1}] \) and \( \text{snat}[A_2] =_{\iota} \text{snat}[\overline{n_2}] \). By inspection of the type equivalence rules in Figure 2.7 we know \( A_1 =_{\beta_{\mu}} \overline{n_1} \) and \( A_2 =_{\beta_{\mu}} \overline{n_2} \). Therefore \( \text{plus } A_1 A_2 =_{\beta_{\mu}} \text{plus } \overline{n_1} \overline{n_2} \). By Lemma A.7, \( \text{plus } A_1 A_2 =_{\beta_{\mu}} \overline{n_1 + n_2} \). Therefore \( \text{snat}[\text{plus } A_1 A_2] =_{\iota} \text{snat}[\overline{n_1 + n_2}] \).

Therefore \( \tau =_{\iota} \text{snat}[\overline{n_1 + n_2}] \).

By the following derivation we have \( \vdash \vdash \overline{n_1 + n_2} : \tau \):

\[
\begin{align*}
\vdash \vdash & \text{Prop : Type} \\
\therefore & \vdash \vdash \overline{n_1} : \text{snat}[A_1] \\
\therefore & \vdash \vdash \overline{n_2} : \text{snat}[A_2] \\
\therefore & \vdash \vdash \overline{n_1 + n_2} : \text{snat}[\overline{n_1 + n_2}] \\
\therefore & \vdash \vdash \overline{n_1 + n_2} : \tau
\end{align*}
\]
(Rule R-LT-T) We require $e \equiv \pi \cdot \pi$ for some $n_1$ and $n_2$, and $n_1 < n_2$. By Lemma A.3, we have $\tau = s_{\text{Boo}}[\text{lt } A_1 A_2]$, $\vdash \pi_1 : s_{\text{Nat}}[A_1]$, and $\vdash \pi_2 : s_{\text{Nat}}[A_2]$, for some $A_1$ and $A_2$. Similar to the proof in the above case, we know that $\text{lt } A_1 A_2 = \beta_0$, $\pi_1 \pi_2$. Since $n_1 < n_2$, by Lemma A.7, $\pi_1 \pi_2 = \beta_0$ true. Therefore $\tau = s_{\text{Boo}}[\text{true}].$

By the rule E-TRUE, we know $\vdash e : s_{\text{Boo}}[\text{true}]$. Therefore, we get $\vdash e : \tau$ by rule E-CONV.

(Rule R-LT-F) The proof is similar to the above case.

(Rule R-if-T) We require $e \equiv \text{if}[B,A](\tau_1,e_1,\tau_2,e_2)$. By Lemma A.3, we have $\vdash \pi : s_{\text{Boo}}[A']$, $\vdash A : B A'$, $\vdash e_1 : \tau'$, $\vdash e_1 : \tau' : \Omega$, and $\tau = \tau'$, for some $A'$ and $\tau'$.

Again by Lemma A.3 we know $s_{\text{Boo}}[A'] = s_{\text{Boo}}[\text{true}]$, therefore $A' = \beta_0$ true. Therefore $\vdash A : B \text{ true}$. By Lemma A.6 we know $\vdash e_1 : A/X_1[e_1] : \tau'$ (we know $X_1$ is not free in $\tau'$ by $\vdash e_1 : \tau' : \Omega$). Since $\tau = \tau'$, by the rule E-CONV we know $\vdash e_1 : \tau_1 : \tau$.

(Rule R-if-F) The proof is similar to the above case.

(Rule R-β) We require $e \equiv (\lambda x : \tau_1.e')v$ for some $x$, $\tau_1$ and $e'$. By Lemma A.3, we have $\vdash \lambda x : \tau_1.e' : \tau_1 \rightarrow \tau_2$, $\vdash v : \tau_1$ and $\tau = \tau_2$ for some $\tau_2$.

Again by Lemma A.3, we have $\vdash \tau_1 : \Omega$, $\vdash x : \tau_1 \vdash e' : \tau_2'$, and $\tau_1 \rightarrow \tau_2 = \tau_1 \rightarrow \tau_2'$.

It is easy to prove that $\tau_2 = \tau_2'$ from $\tau_1 \rightarrow \tau_2 = \tau_1 \rightarrow \tau_2'$. Since $\tau = \tau_2$, we have $\tau = \tau_2'$. By Lemma A.4 we have $\vdash e_1 : e' : \tau_2'$. Then $\vdash e_1 : e' : \tau$ follows the rule E-CONV.

(Rule R-tapp1) The proof is similar to the above case. In this case we use Lemma A.5 instead of Lemma A.4.

(Rule R-tapp2) The proof is similar to the case for rule R-β. In this case we use Lemma A.6 instead of Lemma A.4.

(Rule R-fix) The proof is similar to the case for rule R-β.

(Rule R-sel) We require $e \equiv \{l_1 = v_1, \ldots, l_n = v_n\}.l_i$ for some $\tau_1, \ldots, \tau_n$, and $1 \leq i \leq n$. By Lemma A.3 we know $\vdash \pi : \{l_1 = v_1, \ldots, l_n = v_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ and $\tau = \tau_i$.

Again by Lemma A.3 we know there exist $\tau_1', \ldots, \tau_n'$ such that $\{l_1 : \tau_1', \ldots, l_n : \tau_n'\} = \{l_1 : l_1', \ldots, l_n : l_n'\}$ and for $i = 1, \ldots, n$ we have $\vdash \pi : \tau_i'$.

It is easy to prove that for each $i$ ($1 \leq i \leq n$) we have $\tau_i = \tau_i'$, since $\{l_1 : l_1', \ldots, l_n : l_n'\} = \{l_1 : l_1', \ldots, l_n : l_n'\}$. Since $\tau = \tau_i$, we know $\tau = \tau_i'$

Therefore we have $\vdash \pi : \tau$ by rule E-CONV.
(Rule R-case1) We require \( e \equiv \text{case}(\text{inj}_1^\prime v, e_1, e_2) \) for some \( \tau', v, e_1 \) and \( e_2 \).
By Lemma A.3 we know \( \vdash_1 \vdash \text{inj}_1^\prime v : \tau_1 + \tau_2 \). \( \vdash_1 \vdash e_1 : \tau_1 \rightarrow \tau_0 \), \( \vdash_1 \vdash e_2 : \tau_2 \rightarrow \tau_0 \), and \( \tau =_t \tau_0 \), for some \( \tau_0, \tau_1 \) and \( \tau_2 \).
Again by Lemma A.3 we know there exist \( \tau'_1 \) and \( \tau'_2 \) such that \( \tau' \equiv \tau'_1 + \tau'_2 \).
\( \tau'_1 + \tau'_2 =_t \tau_1 + \tau_2 \), and \( \vdash_1 \vdash v : \tau'_1 \).
It is easy to prove that \( \tau_1 =_t \tau'_1 \), since \( \tau'_1 + \tau'_2 =_t \tau_1 + \tau_2 \).
Therefore we have the following derivation of \( \vdash_1 \vdash e_1 v : \tau \).
\[
\begin{array}{c}
\vdash_1 \vdash e_1 : \tau_1 \rightarrow \tau_0 \\
\vdash_1 \vdash v : \tau_1 \\
\vdash_1 \vdash e_1 v : \tau_0 \\
\vdash_1 \vdash e_1 v : \tau
\end{array}
\]

(Rule R-case2) Similar to the proof of the above case.

(Rule R-unfd) We require \( e \equiv \text{unfold}(\text{fold}^\prime v) \) for some \( \tau' \) and \( v \). By Lemma A.3 we know there exist \( \kappa, \tau_1 \) and \( \tau_2 \) such that \( \vdash_1 \vdash \text{fold}^\prime v : \mu_\kappa(\tau_1, \tau_2) \), and \( \tau =_t \mathcal{K}(\kappa, \tau_1, \tau_2) \).
Again by Lemma A.3 we know there exist \( \kappa', \tau'_1 \) and \( \tau'_2 \) such that \( \tau' \equiv \mu_\kappa(\tau'_1, \tau'_2) \), \( \vdash_1 \vdash \mu_\kappa(\tau'_1, \tau'_2) : \Omega \), \( \mu_\kappa(\tau_1, \tau_2) =_t \mu_\kappa(\tau'_1, \tau'_2) \), and \( \vdash_1 \vdash v : \mathcal{K}(\kappa', \tau'_1, \tau'_2) \).
From the fact that \( \mu_\kappa(\tau_1, \tau_2) =_t \mu_\kappa(\tau'_1, \tau'_2) \), it easy to prove that \( \kappa =_t \kappa' \), \( \tau_1 =_t \tau'_1 \) and \( \tau_2 =_t \tau'_2 \). Therefore we can prove that \( \mathcal{K}(\kappa, \tau_1, \tau_2) =_t \mathcal{K}(\kappa', \tau'_1, \tau'_2) \).
Since \( \tau =_t \mathcal{K}(\kappa, \tau_1, \tau_2) \), we know \( \vdash_1 \vdash v : \tau \).

(Rule R-let) Similar to the proof of the case for the rule R-beta.

(Rule R-tlet1) We require \( e \equiv \text{let } t = \tau_0 \text{ in } e' \). By Lemma A.3 we know there exist \( \kappa \) and \( \tau' \) such that \( \vdash_1 \vdash \tau_0 : \kappa \), \( \vdash_1 \vdash [\tau_0/t]e' : \tau' \), and \( \tau =_t \tau' \).
Therefore we have \( \vdash_1 \vdash [\tau_0/t]e' : \tau \).

(Rule R-tlet2) Similar to the proof of the above case.

(Rule R-open) We require \( e \equiv \text{let } (t, x) = \langle t' = \tau_1, v : \tau_2 \rangle \text{ in } e' \). By Lemma A.3 we know there exist \( \kappa, \tau_0 \) and \( \tau' \) such that \( \vdash_1 \vdash \tau' : \Omega \), \( \vdash_1 \vdash \langle t' = \tau_1, v : \tau_2 \rangle : \exists t : \kappa, \tau_0 \vdash_1 \vdash t : \kappa, x : \tau_0 \vdash_1 \vdash e' : \tau' \), and \( \tau =_t \tau' \).
Again by Lemma A.3 we know there exists \( \kappa' \) such that \( \vdash_1 \vdash \tau_1 : \kappa' \), \( \vdash_1 \vdash v : [\tau_1/t] \tau_2 \), and \( \exists t : \kappa, \tau_0 =_t \exists t' : \kappa', \tau_2 \).
It is easy to prove that \( \kappa' \equiv \kappa \) and \( \tau_2 =_t [t'/t] \tau_0 \). By Proposition 2.6 we know \( [\tau_1/t'] \tau_2 =_t [\tau_1/t'] [t'/t] \tau_0 \), i.e. \( [\tau_1/t'] \tau_2 =_t [\tau_1/t] \tau_0 \). By the typing rule E-conv we know \( \vdash_1 \vdash v : [\tau_1/t] \tau_0 \).

By Lemma A.5 and A.4 we know \( \vdash_1 \vdash [v/x][\tau_1/t] e : [\tau_1/t] \tau' \). Since we have \( \vdash_1 \vdash \tau' : \Omega \), \( t \) must not be free in \( \tau' \). Therefore \( \vdash_1 \vdash [v/x][\tau_1/t] e : \tau' \). Again by typing rule E-conv we finally get \( \vdash_1 \vdash [v/x][\tau_1/t] e : \tau \).
(Rule \textbf{R-open2}) Similar to the proof for the above case. In this case we use Lemma A.6 instead of Lemma A.5.

\textbf{Theorem A.9 (Type Preservation)} If \(\cdot;\cdot;\vdash e : \tau\) and \(e \mapsto^* e'\), then \(\cdot;\cdot;\vdash e' : \tau\).

\textbf{Proof.} Since evaluation contexts binds no variables, by Lemma A.8 and A.4, we have if \(\cdot;\cdot;\vdash e : \tau\) and \(e \mapsto e'\), then \(\cdot;\cdot;\vdash e' : \tau\). Then the result follows by induction on the length of the reduction sequence \(e \mapsto^* e'\).

\textbf{Theorem A.10 (Safety of CL)} If \(\cdot;\cdot;\vdash e : \tau\), then either \(e \mapsto^* v\) and \(\cdot;\cdot;\vdash v : \tau\), for some \(v\), or \(e\) diverges (i.e., for each \(e'\), if \(e \mapsto^* e'\), then there exists \(e''\) such that \(e' \mapsto e''\)).

\textbf{Proof.} Follows Theorems A.2 and A.9.