STUDY ON PROOF OF TYPE PRESERVATION IN CPS TRANSFORMATION
CPS 変換における型保存の証明に関する研究

by

Yuki Watanabe

渡邊 裕貴

A Master Thesis

修士論文

Submitted to

the Department of Computer Science

the Graduate School of Information Science and Technology

the University of Tokyo

on February 8, 2011

in partial fulfillment of requirements

for the Degree of Master of Information Science and Technology

Thesis supervisor: Akinori Yonezawa 米澤 明憲

Professor of Computer Science

ABSTRACT

Approaches to proving type preservation property of call-by-value CPS transformation are investigated for various type systems and formalization methods. Firstly, Coq proof scripts of simple type preservation are compared between four formalization methods: the unsorted named, two-sorted named, two-sorted de Bruijn, and two-sorted locally nameless representations. It is shown that two-sorted representations make proofs of type preservation simpler and that the locally nameless representation is more suitable than the other representations when term substitution is involved in language definition and proofs. Next, a type system with singleton types and subtyping is defined so that the semantics of CPS-transformed terms can be denoted using singleton types. A proof sketch of type preservation is presented for the target language typed with singleton types. Lastly, it is shown that proving the preservation of dependent types in the call-by-value CPS transformation is more difficult because it requires a proof of term equivalence that is not required in the case of the call-by-name CPS transformation. Conveying the semantics of terms with singleton types is proposed to show the required equivalence.

論文要旨

本論文では call-by-value の CPS 変換における型の保存を証明するためのアプローチを、様々な型システムや形式化手法に対して追究する。初めに、単純な型の保存の Coq での証明に関して四つの形式化手法 (the unsorted named, two-sorted named, two-sorted de Bruijn, and two-sorted locally nameless representations) を比較する。Two-sorted な手法では証明が簡単になり、また項の置換を取り扱う際には the locally nameless representation がより適していることを示す。続いて、シングルトン型およびサブタイピングを含む型システムを定義し、シングルトン型を用いて CPS 変換後の項の意味論を表す。そして変換後の項がシングルトン型で型付けされるような型の保存を示すための証明の概略を示す。最後に、call-by-value の CPS 変換で依存型の保存を示すためには、call-by-name の場合には不要な等価関係を導く必要があり、そのために証明が難しくなっていることを示す。その等価関係を導くためにシングルトン型を使って項の意味論を伝達するアプローチを提案する。

ACKNOWLEDGMENT

I thank Toshiyuki Maeda for useful discussions about type systems. I am also grateful to Akinori Yonezawa for helpful guidance.

CONTENTS

LIST OF FIGURES

1 Introduction

Compilers are important programs in the sense that many other programs, or even compilers themselves, depend on compilers to produce executable code from source code. For the executable code to work as expected, the compiler must produce code that behaves in the same way as the source code. In other words, the semantics of the program must be preserved between the source and executable code.

Much research has been done to ensure that compilers preserve the semantics of code [8, 10, 18, 20, 25]. One of the standard ways to do that is to formally define the transformation algorithm of the compiler from the source to the target language and then prove that any code transformed by the algorithm has the equivalent semantics in the source and the target language. Because such a formalization and proof are often complex and prone to human errors, it is becoming more common to use proof assistant software such as Coq [7, 12].

Although using a proof assistant ensures strict correctness of the proof, it requires the whole proof to be written in a way that can be accepted by the predefined axioms of the proof assistant. Therefore, when proving semantics preservation using a proof assistant, we sometimes need to take an approach different from when doing a manual proof. Especially, the definitions of the language syntax and semantics should be so formalized that conditions that appear in proofs can be represented by simple expressions. Otherwise, the formalization and the proof will be much more complex, tricky, and hard to follow than in a manual proof.

Alpha equivalence in lambda calculi is an instance of such a problem. In a manual proof, typically, alpha equivalence between lambda terms is taken as implicit; alpha-equivalent terms are considered syntactically equivalent and freely exchangeable with each other. In a proof using Coq, however, alpha equivalence cannot be treated implicitly because the logic used in Coq does not allow extending the definition of syntactic equivalence so that alpha-equivalent terms can be treated as if syntactically equivalent. Consequently, we need during the proof to maintain a property that shows alpha-equivalent terms can be safely considered syntactically equivalent.

To overcome this difficulty, it has been proposed to formalize the syntax of lambda calculi in special ways. In the de Bruijn index representation [16], variables are represented as natural numbers that identify the corresponding variable bindings, making alpha-equivalent terms syntactically equivalent. In the locally nameless representation [2, 9, 19], both usual named variables and de Bruijn indices are used to make the formalization of term substitution easier while maintaining the nice properties of the de Bruijn index representation. In the two-sorted representation [13], two namespaces are used to distinguish variables introduced in transformation from those that have existed since before the transformation.

In this thesis, we investigate and develop type-based verification techniques for CPS transformation of simply and dependently typed lambda calculi. CPS transformation [31, 35] is part of a typical compilation algorithm for a functional language. We verify CPS transformation algorithms by showing that any well-typed term is transformed into a well-typed term.

This thesis chiefly makes three contributions. The first contribution is comparison between various ways of language formalization in the Coq proof assistant. It is shown how the simply typed lambda calculus and CPS transformation algorithms can be formally defined in Coq using each of the unsorted named, two-sorted named, two-sorted de Bruijn index, and two-sorted locally nameless representations. We also examine how their differences affect the Coq proof of the type preservation property of CPS transformation for simply typed source and target language.

The second contribution is that a typed lambda calculus with singleton types [3, 4, 24, 34] is introduced as the target language of CPS transformation. Singleton types are used to denote the semantics of transformed terms. A proof sketch is presented to show type preservation of the CPS transformation from the simply typed lambda calculus to the target language typed with singleton types.

The third contribution is an approach to showing type equivalence that is necessary to prove the type preservation property of the call-by-value CPS transformation on a dependently typed lambda calculus. Dependent types [5, 17, 37] allow typing with more specific information about terms than simple types do. To compile a dependently typed language to another dependently typed language, CPS transformation must preserve the dependent typing. However, proving dependent type preservation of the call-by-value CPS transformation is much harder than that of the call-by-name CPS transformation because we need to show equivalence that is not needed in the cases of simple type preservation or the call-by-name CPS transformation. In this thesis, using singleton types is proposed to show the equivalence needed. The author conjectures that the approach will make it possible to prove dependent type preservation of the call-by-value CPS transformation.

2 Background: CPS Transformation

CPS transformation is a procedure that transforms a lambda term into the continuation-passing style (CPS) [35]. In CPS, a function call never returns: a function is always called with a continuation function given as an argument and then, instead of returning the result to the caller, it calls the continuation giving the result as the argument. After CPS transformation, the program is nearer to the assembly language in that the order of term evaluation is explicit. Especially, return from a function is represented as a call to the continuation.

Before defining CPS transformation, we define the syntax of a lambda calculus:

$$t \coloneqq x \mid \text{unit} \mid \lambda x.\, t \mid t\ t$$

The definition is much like the pure lambda calculus, but we have the unit constant value so that the unit type, which is defined later, is inhabited. For now, we use this definition for both the source and target languages.

On the basis of the definition above, the CPS conversion algorithm is defined as a function $[\![\cdot]\!]$ that maps source language terms to target language terms:

$$\begin{aligned}
[\![x]\!] &\coloneqq \lambda k.\, k\, x \\
[\![\text{unit}]\!] &\coloneqq \lambda k.\, k\ \text{unit} \\
[\![\lambda x.\, t]\!] &\coloneqq \lambda k.\, k(\lambda x.\, [\![t]\!]) \\
[\![t_f\ t_a]\!] &\coloneqq \lambda k.\, [\![t_f]\!] \left(\lambda v_f.\, [\![t_a]\!](\lambda v_a.\, v_f\ v_a\ k)\right)
\end{aligned}$$

$$
\begin{array}{rl}
\text{Term} & t \coloneqq x \mid \text{unit} \mid \lambda x.\, t \mid t\ t \\
\text{Source type} & T \coloneqq \text{Unit} \mid T \to T \\
\text{Target type} & T \coloneqq \text{Unit} \mid T \to T \mid \bot \\
\text{Typing Context} & \Gamma \coloneqq \emptyset \mid \Gamma, x{:}\, T
\end{array}
$$

$$
\frac{x{:}\, T \in \Gamma}{\Gamma \vdash x : T}\text{var} \qquad \frac{\Gamma, x{:}\, T_A \vdash t : T_R}{\Gamma \vdash \lambda x.\, t : T_A \to T_R}\text{abs}
$$

$$
\frac{}{\Gamma \vdash \text{unit} : \text{Unit}}\text{unit} \qquad \frac{\Gamma \vdash t_f : T_A \to T_R \quad \Gamma \vdash t_a : T_A}{\Gamma \vdash t_f\ t_a : T_R}\text{app}
$$

Figure 1: Simply typed lambda calculus

The definition simply follows Plotkin's well-known algorithm that transforms a lambda term into the call-by-value CPS [31]. Any transformed term is a function that takes as the argument a continuation function that receives the evaluation result. The variable term x is transformed into a function that, when called with a continuation, calls the continuation with the variable value passed to the continuation. The transformation of the unit value is analogous to that of a variable. That of a lambda abstraction $\lambda x.\, t$ is also similar, but subterm t is recursively transformed. For the function application term $t_1\, t_2$, the transformed term is a function that, given continuation k, evaluates the term $[\![t_1]\!]\big(\lambda v_1.\, [\![t_2]\!](\lambda v_2.\, v_1\, v_2\, k)\big)$, that is, passes the continuation $\big(\lambda v_1.\, [\![t_2]\!](\lambda v_2.\, v_1\, v_2\, k)\big)$ to the transformation of term t_1. As a result, this continuation is called with the evaluation result of term t_1 and evaluates the term $[\![t_2]\!](\lambda v_2.\, v_1\, v_2\, k)$. Likewise, the continuation $(\lambda v_2.\, v_1\, v_2\, k)$ is called with the result of term t_2. Finally, function application is performed in the evaluation of term $v_1\, v_2\, k$. Note that variable v_1 and v_2 are bound to the function and the argument values, respectively. The function is given continuation k as well as the original argument and this continuation receives the result of the function application.

3 Simple Type Preservation and Coq Formalization Methods

In this section, we define a CPS transformation algorithm for the simply typed lambda calculus, then examine and compare four types of formalizations to prove type preservation property of the transformation using the Coq proof assistant. We use the well-known definition of the simply typed lambda calculus (Figure 1) for both the source and target languages.

The term syntax is defined as in Section 2. The function type is represented by an arrow as usual. The unit type is the type of the unit value. In addition, the target language has the bottom type that is the type of imaginary terms returned by continuation functions. A typing context is defined as a finite partial function that maps variable names to types. The typing rules are defined in the standard manner. Note that the only difference between the source and target languages is that the target language has the bottom type. There is no typing rule for the bottom type because no term has the bottom type. Since we are focusing on the proof of type preservation in this thesis, we simply assume the soundness of the type system and do not elaborate on the proof of the soundness, which should be found in any textbook on typed lambda calculi [21, 29].

We now define the CPS transformation of types and contexts:

$$\begin{aligned} [\![\text{Unit}]\!] &\coloneqq \text{Unit} \\ [\![T_A \to T_R]\!] &\coloneqq [\![T_A]\!] \to ([\![T_R]\!] \to \bot) \to \bot \end{aligned}$$

$$\begin{aligned} [\![\emptyset]\!] &\coloneqq \emptyset \\ [\![\Gamma, x{:}\, T]\!] &\coloneqq [\![\Gamma]\!], x{:}\, [\![T]\!] \end{aligned}$$

The unit type is transformed into itself. For a function type, the types of the argument and the result are recursively transformed and the result type becomes the argument type of the continuation that receives the result of the function. A typing context is transformed by simply transforming the types in it.

The type preservation property states that, if a term in the source language is well-typed, the transformation of the term in the target language is well-typed as well.

Theorem 1: (Type preservation) If the typing judgment $\Gamma \vdash t : T$ is derivable in the source language, then $[\![\Gamma]\!] \vdash [\![t]\!] : ([\![T]\!] \to \bot) \to \bot$ is derivable in the target language.

In the rest of this section, we look into how this theorem can be proved using the Coq proof assistant with various formalizations of the source and target language syntaxes, typing rules, and transformation algorithms.

For brevity, $T \to \bot$ is abbreviated as $\neg T$ in the rest of this thesis. Note that $\neg\neg T$ is $(T \to \bot) \to \bot$.

3.1 Named Representations

In the CPS transformation of a term, a new lambda abstraction is introduced so that the converted term can receive a continuation. For the transformation of a function term, two more lambda abstractions are introduced that are passed as continuations to the recursively transformed subterms. Then, what are the names of the variables bound in these new lambda abstractions? The transformed term may have an undesired semantics if a variable name that is already bound elsewhere is bound again in the new lambda abstractions. To avoid this problem, the variable names for the new lambda abstractions must be *fresh*.

In the next two subsections, we introduce two formalization methods, that is, the unsorted named representation and the two-sorted named representation, and see that the latter eases the burden in proving the type preservation property. These methods differ in the way of choosing variable names for lambda abstractions newly introduced in the transformation.

In the two methods, variables are distinguished by *names* as in usual lambda calculi (and hence they are called “named representations”). The set of names may be any infinite set as long as the equality of any two elements of the set is decidable. We use the set of natural numbers as the set of names in proofs using Coq because the proof of the decidability is provided in Coq’s built-in library.

3.1.1 Unsorted Named Representation

In the unsorted named representation, the names of new variables are carefully chosen so that the new variables do not override existing variable bindings. We compute the set of free variables of the (sub)term and choose a fresh name that is not in the set. The set of the free variables of a term is computed as follows:

$$\begin{aligned}
\mathrm{FV}(x) &\coloneqq \{x\} \\
\mathrm{FV}(\mathrm{unit}) &\coloneqq \emptyset \\
\mathrm{FV}(\lambda x.\, t) &\coloneqq \mathrm{FV}(t) \setminus \{x\} \\
\mathrm{FV}\big(t_f\ t_a\big) &\coloneqq \mathrm{FV}\big(t_f\big) \cup \mathrm{FV}(t_a)
\end{aligned}$$

We use an (ordered) list of natural numbers to represent a set of variables in Coq. Given a list of natural numbers, we can yield a fresh name by calculating the maximum number in the list plus one.

Now we redefine the term transformation algorithm with the choice of new variable names taken into account:

$$\begin{array}{rll}
[\![x]\!] &:= \lambda k.\, k\, x & k \notin \mathrm{FV}(x) \\
[\![\mathrm{unit}]\!] &:= \lambda k.\, k\, \mathrm{unit} & k \notin \mathrm{FV}(\mathrm{unit}) \\
[\![\lambda x.\, t]\!] &:= \lambda k.\, k(\lambda x.\, [\![t]\!]) & k \notin \mathrm{FV}(\lambda x.\, t) \\
[\![t_f\; t_a]\!] &:= \lambda k.\, [\![t_f]\!]\left(\lambda v_f.\, [\![t_a]\!](\lambda v_a.\, v_f\; v_a\; k)\right) & k \notin \mathrm{FV}(t_f\; t_a), v_f \notin \{k\} \cup \mathrm{FV}(t_a), v_a \notin \{k, v_f\}
\end{array}$$

We do not care about variables that do not occur free in the (sub)term. Such variables can be safely re-bound even if they are bound outside the term. To allow re-binding of variables, we assume that the mapping from names to types in a typing context can be redefined. For example, if $x{:}\,T_1 \in \Gamma$, then $x{:}\,T_1 \notin \Gamma, x{:}\,T_2$ unless $T_1 = T_2$.

The statement of the type preservation property shown above (Theorem 1) is actually too strong to prove in the current named representation setting. Since new lambda abstractions are introduced in the transformation, the variables bound in those abstractions, which has no corresponding counterparts in the source term, are added to the typing context when typing subterms of a transformed term. The statement of the type preservation have to be so weakened as to allow existence of such variables in the typing context of the target language (otherwise, we cannot appeal to the induction hypothesis to show that a subterm is well-typed). We define an auxiliary predicate that states that, for each variable in a specific list of variables, if the variable is mapped to a type in the source typing context, then the variable is mapped to the corresponding type in the target typing context:

$$\mathrm{CC}(\Gamma; \Gamma'; l) := \forall x \in l, \forall T, x{:}\,T \in \Gamma \Rightarrow x{:}\,[\![T]\!] \in \Gamma'$$

Using this predicate, the type preservation property is restated:

Theorem 2: (Type preservation) If the typing judgment $\Gamma \vdash t : T$ is derivable in the source language and $\mathrm{CC}(\Gamma; \Gamma';\ \mathrm{FV}(t))$ holds, then $\Gamma' \vdash [\![t]\!] : \neg\neg[\![T]\!]$ is derivable in the target language.

$$\begin{array}{rl}\text{Term} & t \coloneqq x \mid x' \mid \text{unit} \mid \lambda x.t \mid \lambda x'.t \mid t\,t \\ \text{Type} & T \coloneqq \text{Unit} \mid T \to T \mid \bot \\ \text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, x{:}T\end{array}$$

$$\frac{x{:}T \in \Gamma}{\Gamma;\Gamma' \vdash x : T}\,\text{var} \qquad \frac{\Gamma, x{:}T_A;\Gamma' \vdash t : T_R}{\Gamma;\Gamma' \vdash \lambda x.t : T_A \to T_R}\,\text{abs}$$

$$\frac{x{:}T \in \Gamma'}{\Gamma;\Gamma' \vdash x' : T}\,\text{var'} \qquad \frac{\Gamma;\Gamma', x{:}T_A \vdash t : T_R}{\Gamma;\Gamma' \vdash \lambda x'.t : T_A \to T_R}\,\text{abs'}$$

$$\frac{}{\Gamma;\Gamma' \vdash \text{unit} : \text{Unit}}\,\text{unit} \qquad \frac{\Gamma;\Gamma' \vdash t_f : T_A \to T_R \quad \Gamma;\Gamma' \vdash t_a : T_A}{\Gamma;\Gamma' \vdash t_f\,t_a : T_R}\,\text{app}$$

Figure 2: Target language in the two-sorted named representation

This theorem can be proved in the Coq proof assistant by induction on the derivation of the source typing judgment. The proof is straightforward; we sketch only the case of the abs rule here.

In the case of the abs rule, where $t = \lambda x.t'$ and $T = T_A \to T_R$ and $\Gamma, x{:}T_A \vdash t' : T_R$, the goal is to derive

$$\Gamma' \vdash \lambda k.k(\lambda x.[\![t']\!]) : \neg\neg([\![T_A]\!] \to \neg\neg[\![T_R]\!]).$$

By applying applicable rules as follows, the goal is reduced to showing $\Gamma', k{:}T_k, x{:}[\![T_A]\!] \vdash [\![t]\!] : \neg\neg[\![T_R]\!]$ where $T_k \coloneqq \neg([\![T_A]\!] \to \neg\neg[\![T_R]\!])$:

$$\cfrac{\cfrac{\cfrac{k{:}T_k \in \Gamma', k{:}T_k}{\Gamma', k{:}T_k \vdash k{:}T_k}\,\text{var} \qquad \cfrac{\Gamma', k{:}T_k, x{:}[\![T_A]\!] \vdash [\![t']\!] : \neg\neg[\![T_R]\!]}{\Gamma', k{:}T_k \vdash \lambda x.[\![t']\!] : [\![T_A]\!] \to \neg\neg[\![T_R]\!]}\,\text{abs}}{\Gamma', k{:}T_k \vdash k(\lambda x.[\![t']\!]) : \bot}\,\text{app}}{\Gamma' \vdash \lambda k.k(\lambda x.[\![t']\!]) : \neg T_k}\,\text{abs}$$

By the definition of transformation algorithm, variable k is either x or not a free variable of $[\![t']\!]$ and thus we have $\text{CC}(\Gamma, x{:}T_A; \Gamma', k{:}\ldots, x{:}[\![T_A]\!]; \text{FV}(t))$. This enables applying the induction hypothesis to $\Gamma, x{:}T_A \vdash t' : T_R$ to obtain $\Gamma', k{:}T_k, x{:}[\![T_A]\!] \vdash [\![t']\!] : \neg\neg[\![T_R]\!]$.

3.1.2 Two-Sorted Named Representation

In the two-sorted named representation, variables in the target language are categorized into two namespaces: one is for variables that appear in source terms and the other for variables that are introduced in the transformation. The names of variables introduced in the transformation never override variable bindings from source terms because they are in the different namespaces. As a result, the names of new variables can be arbitrarily chosen without care about conflict with existing names. This idea of using two namespaces in the target language originated with Dargaye and Leroy [13].

Now we redefine the target language in the two-sorted manner as shown in Figure 2. The syntax of terms contains two forms of the variable term and the lambda abstraction term. Variables without a prime are used for variables that come from source terms and variables with a prime are for variables introduced in the transformation. The syntax of types and typing contexts are the same as in the unsorted representation. The typing judgment is of the new form $\Gamma; \Gamma' \vdash t : T$, which contains two typing contexts. The first context is for variables without a prime and the second for variables with a prime. The typing rules of a variable and a lambda abstraction term correspondingly have two forms.

The transformation algorithm for terms is redefined as follows (the definition of the source language is just the same as in the previous unsorted representation):

$$\begin{aligned}
[\![x]\!] &\coloneqq \lambda k'.\, k'\, x \\
[\![\text{unit}]\!] &\coloneqq \lambda k'.\, k'\, \text{unit} \\
[\![\lambda x.\, t]\!] &\coloneqq \lambda k'.\, k'(\lambda x.\, [\![t]\!]) \\
[\![t_f\, t_a]\!] &\coloneqq \lambda k'.\, [\![t_f]\!] \left(\lambda v_f'.\, [\![t_a]\!](\lambda v_a'.\, v_f'\, v_a'\, k')\right) \\
&\quad \text{where } k', v_f', \text{ and } v_a' \text{ are mutially different}
\end{aligned}$$

Variables that appear in the source term are simply transformed into a variable with the same name without a prime. Variables that are introduced in the transformation are all with a prime.

Before proving the type preservation property, we introduce an auxiliary lemma:

Lemma 3: If $x{:}T \in \Gamma$, then $x{:}[\![T]\!] \in [\![\Gamma]\!]$.

In the Coq formalization of the named representation, the relation $x{:}T \in \Gamma$ is defined as:

$$\begin{aligned}
x{:}T \in \emptyset &\coloneqq \bot \\
x{:}T \in \Gamma, x'{:}T' &\coloneqq x = x' \land T = T' \lor x \neq x' \land x{:}T \in \Gamma
\end{aligned}$$

The proof of Lemma 3 is by straightforward induction on the structure of the ordered list representing the typing context.

Theorem 4: (Type preservation) If the typing judgment $\Gamma \vdash t : T$ is derivable in the source language, then $[\![\Gamma]\!]; \Gamma' \vdash [\![t]\!] : \neg\neg[\![T]\!]$ is derivable in the target language.

The proof, similar to that of Theorem 2, is by induction on the derivation of the source typing judgment. We sketch the case of the var and abs rules here.

- In the case of the var rule, where $t = x$ and $x{:}T \in \Gamma$, the goal is to derive

$$[\![\Gamma]\!]; \Gamma' \vdash \lambda k'.\, k'\, x : \neg\neg[\![T]\!].$$

By applying applicable rules as follows, the goal is reduced to showing $x{:}\,[\![T]\!] \in [\![\Gamma]\!]$:

$$\dfrac{\dfrac{\dfrac{k{:}\,\neg[\![T]\!] \in \Gamma', k{:}\,\neg[\![T]\!]}{[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash k' : \neg[\![T]\!]}\,\text{var} \qquad \dfrac{x{:}\,[\![T]\!] \in [\![\Gamma]\!]}{[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash x : [\![T]\!]}\,\text{var}}{[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash k'\, x : \bot}\,\text{app}}{[\![\Gamma]\!]; \Gamma' \vdash \lambda k'.\, k'\, x : \neg\neg[\![T]\!]}\,\text{abs}$$

Applying Lemma 3 to $x{:}T \in \Gamma$ gives $x{:}\,[\![T]\!] \in [\![\Gamma]\!]$.

- In the case of the abs rule, where $t = \lambda x.\, t'$ and $T = T_A \to T_R$ and $\Gamma, x{:}\,T_A \vdash t' : T_R$, the goal is to derive

$$[\![\Gamma]\!]; \Gamma' \vdash \lambda k'.\, k'(\lambda x.\, [\![t']\!]) : \neg\neg([\![T_A]\!] \to \neg\neg[\![T_R]\!]).$$

By applying applicable rules, the goal is reduced to $[\![\Gamma]\!], x{:}\,[\![T_A]\!]; \Gamma', k{:}\,T_k \vdash [\![t']\!] : \neg\neg[\![T_R]\!]$. This is equivalent to $[\![\Gamma, x{:}\,T_A]\!]; \Gamma', k{:}\,T_k \vdash [\![t']\!] : \neg\neg[\![T_R]\!]$ because $[\![\Gamma]\!], x{:}\,[\![T_A]\!] = [\![\Gamma, x{:}\,T_A]\!]$ by definition. This goal is sufficed by applying the induction hypothesis to $\Gamma, x{:}\,T_A \vdash t' : T_R$.

This proof is more simplified than that of Theorem 2 because we do not have to prove a condition about the CC predicate before applying the induction hypothesis.

3.2 Indexed Representations

Lambda terms that differ only in the choice of bound variable names are alpha-equivalent terms. For example, the terms $\lambda x.\lambda y.\, x\, y\, z$, $\lambda y.\lambda x.\, y\, x\, z$, and $\lambda a.\lambda b.\, a\, b\, z$ are all alpha-equivalent to each other. Alpha-equivalent terms are often considered substitutable with each other in manual proofs because those terms have the same semantics. In proofs using the Coq proof assistant, however, equivalence and substitutability of such terms must be explicitly formalized and treated, which makes the statements of theorems more redundant and proofs more complicated. To overcome this difficulty, in the following subsections, we introduce two

$$\begin{array}{rl} \text{Term} & t \coloneqq n \mid \text{unit} \mid \lambda.\, t \mid t\ t \\ \text{Type} & T \coloneqq \text{Unit} \mid T \to T \\ \text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, T \end{array}$$

$$\frac{\Gamma(n) = T}{\Gamma \vdash n : T}\text{var} \qquad \frac{\Gamma, T_A \vdash t : T_R}{\Gamma \vdash \lambda.\, t : T_A \to T_R}\text{abs}$$

$$\frac{}{\Gamma \vdash \text{unit} : \text{Unit}}\text{unit} \qquad \frac{\Gamma \vdash t_f : T_A \to T_R \quad \Gamma \vdash t_a : T_A}{\Gamma \vdash t_f\ t_a : T_R}\text{app}$$

Figure 3: Source language in the de Bruijn index representation

types of formalization of lambda calculus in which alpha-equivalent terms are all syntactically equivalent. We also prove the type preservation property in each of the two formalizations as well.

3.2.1 Two-Sorted de Bruijn Index

The formalization introduced in this subsection is the two-sorted version of the *de Bruijn index* representation. The de Bruijn index representation, named after its inventor, uses natural numbers instead of names to represent variables [16]. No explicit name is bound in a lambda abstraction and each variable that occurs in a term is distinguished by its number that indicates the corresponding binding. The named term $\lambda x. x(\lambda y. y\ x)$, for example, is represented as $\lambda. 0(\lambda. 0\ 1)$ using de Bruijn indices. Variable 0 corresponds to the binding by the innermost lambda abstraction (relative to the variable occurrence) and Variable 1 to the second innermost abstraction, and so on. If there is no abstraction that corresponds to a variable's number, the variable is considered a free variable. For example, the named term $x(\lambda y. x)$ is represented as $0(\lambda. 1)$ with de Bruijn indices on the assumption that variable x corresponds to the variable first bound outside the term.

We first redefine the source language using the de Bruijn index representation (Figure 3). Note that the variable term is represented by a natural number instead of a name and that the lambda abstraction term has no name to bind. The typing context is defined as a finite ordered list of types. We write the nth element of list Γ as $\Gamma(n)$. Variable n is well-typed if $\Gamma(n)$ is the

$$\begin{array}{rl} \text{Term} & t \coloneqq n \mid n' \mid \text{unit} \mid \lambda.\, t \mid \lambda'.\, t \mid t\ t \\ \text{Type} & T \coloneqq \text{Unit} \mid T \to T \mid \bot \\ \text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, T \end{array}$$

$$\frac{\Gamma(n) = T}{\Gamma; \Gamma' \vdash n : T}\text{var} \qquad \frac{\Gamma, T_A; \Gamma' \vdash t : T_R}{\Gamma; \Gamma' \vdash \lambda.\, t : T_A \to T_R}\text{abs}$$

$$\frac{\Gamma'(n) = T}{\Gamma; \Gamma' \vdash n' : T}\text{var'} \qquad \frac{\Gamma; \Gamma', T_A \vdash t : T_R}{\Gamma; \Gamma' \vdash \lambda'.\, t : T_A \to T_R}\text{abs'}$$

$$\frac{}{\Gamma; \Gamma' \vdash \text{unit} : \text{Unit}}\text{unit} \qquad \frac{\Gamma; \Gamma' \vdash t_f : T_A \to T_R \quad \Gamma; \Gamma' \vdash t_a : T_A}{\Gamma; \Gamma' \vdash t_f\ t_a : T_R}\text{app}$$

Figure 4: Target language in the two-sorted de Bruijn index

type of the variable (the var rule) because the types of free variables are stored in the context in the order the variables are bound (the abs rule).

The new formalization of the target language combines the two-sorted representation, introduced in the previous section, with the de Bruijn index representation (Figure 4). The new definition of the syntax and typing rules resemble those of the two-sorted named representation except that the new definition uses the de Bruijn index representation.

The transformation algorithm of terms and typing contexts are also redefined according to the de Bruijn index representation:

$$\begin{aligned} [\![n]\!] &\coloneqq \lambda'.\, 0'\ n \\ [\![\text{unit}]\!] &\coloneqq \lambda'.\, 0'\ \text{unit} \\ [\![\lambda.\, t]\!] &\coloneqq \lambda'.\, 0'(\lambda.\, [\![t]\!]) \\ [\![t_f\ t_a]\!] &\coloneqq \lambda'.\, [\![t_f]\!]\big(\lambda'.\, [\![t_a]\!](\lambda'.\, 1'\ 0'\ 2')\big) \end{aligned}$$

$$\begin{aligned} [\![\emptyset]\!] &\coloneqq \emptyset \\ [\![\Gamma, T]\!] &\coloneqq [\![\Gamma]\!], [\![T]\!] \end{aligned}$$

The proof of the type preservation property in the two-sorted de Bruijn representation proceeds in the same way as in the two-sorted named representation. We first prove a lemma corresponding to Lemma 3 used in the two-sorted named representation:

Lemma 5: If $\Gamma(n) = T$, then $[\![\Gamma]\!](n) = [\![T]\!]$.

The proof is by a straightforward induction on the structure of the ordered list representing the typing context.

Theorem 6: (Type preservation) If the typing judgment $\Gamma \vdash t : T$ is derivable in the source language, then $[\![\Gamma]\!]; \Gamma' \vdash [\![t]\!] : \neg\neg[\![T]\!]$ is derivable in the target language.

The proof is by induction on the source typing derivation, similar to the proof of Theorem 4.

In the named representations, alpha conversion may be required to avoid variable captures in capture-avoiding substitution of a term. For example, the substitution of free variable x in the term $(\lambda y. y\, x)$ with $(\lambda w. y)$ yields $(\lambda z. z(\lambda w. y))$ where $z \neq y$, in which bound variable y has been renamed to z. Such renaming makes a proof that contains term substitution more complicated because during the proof we need to track which variables are renamed, ensure that alpha conversion is correctly done, etc. Moreover, the statement of the theorem being proved often need to be carefully specified so that the induction hypothesis can be applied to conditions that contain alpha-converted terms in an inductive proof of the theorem.

In the de Bruijn index representation, such proof burden is eased because alpha-equivalent terms are always syntactically equivalent [16]. However, the numbers that represent free variables need to be adjusted during substitution so that the numbers continue to denote the same variables as before the substitution. For example, substituting the free variable in the term $(\lambda. 0\, 1)$ to the term $(\lambda. 2)$ results in $(\lambda. 0(\lambda. 3))$, in which free variable 2 has been incremented to 3 because the substituted free variable was in a lambda abstraction. In a proof that contains term substitution, we need to ensure that variables are correctly adjusted during substitution, track which numbers denote the same free variable, etc. As a result, we encounter another difficulty in proving theorems in the de Bruijn index representation.

Using the de Bruijn index representation is a bad idea also if variables appear in types in the language involved. An example of such a language is the dependently typed lambda calculus described in Section 5. In the dependently typed lambda calculus, typing such as

$$n{:}\mathrm{Nat}, l{:}\,\mathrm{List}\, n \vdash l : \mathrm{List}\, n$$

is possible. Note that variable n appears in the type $\mathrm{List}\, n$. With de Bruijn indices, this typing judgment would be represented as

$$\mathrm{Nat}, \mathrm{List}\,0 \vdash 0 : \mathrm{List}\,1\,,$$

where the same two types $\mathrm{List}\,0$ and $\mathrm{List}\,1$ have different syntactic representations. To type-check this judgment, we need to check if these two types are the same type, but the comparison of the types is much more complicated when the same two types are syntactically different.

A major cause of the drawbacks of the de Bruijn index representation is that the number representing a free variable in a term depends on the context in which the variable appears; the same free variables in a term may be represented by different numbers. As a result, free variables' number need to be adjusted when comparison, substitution, etc. happen in a different context. The locally nameless representation, introduced in the next subsection, solves this problem by using names for free variables while using de Bruijn indices for bound variables.

3.2.2 Two-Sorted Locally Nameless Representation

In the locally nameless representation [2, 9, 19], free variables are represented by names as in the usual named representation and bound variables are represented by de Bruijn indices. The locally nameless representation inherits the nice property of the de Bruijn index representation: alpha-equivalent terms are always syntactically equivalent. At the same time, the same free variables have the same representation because they are represented by names.

In the locally nameless representation, term substitution can be done by simply replacing free variables in a term with another term. Renaming bound variables is not required because they are not represented by names unlike the named representation. Adjusting free variables' numbers is not required either because they are distinguished by names rather than de Bruijn indices. For example, the substitution of free variable x in the term $(\lambda.\,0\,x)$ with $(\lambda.\,y)$ yields $(\lambda.\,0(\lambda.\,y))$.

The definition of the source language in the locally nameless representation is straightforward:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\text{var} \qquad \frac{\forall x \notin l \quad \Gamma, x : T_A \vdash t^x : T_R}{\Gamma \vdash \lambda. t : T_A \to T_R}\text{abs}$$

$$\frac{}{\Gamma \vdash \text{unit} : \text{Unit}}\text{unit} \qquad \frac{\Gamma \vdash t_f : T_A \to T_R \quad \Gamma \vdash t_a : T_A}{\Gamma \vdash t_f\, t_a : T_R}\text{app}$$

Figure 5: Typing rules of the source language in the locally nameless representation

$$\begin{array}{rl} \text{Term} & t \coloneqq n \mid x \mid \text{unit} \mid \lambda. t \mid t\ t \\ \text{Type} & T \coloneqq \text{Unit} \mid T \to T \\ \text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, x : T \end{array}$$

The term syntax contains both the bound variable term n and the free bound variable term x. A typing context is a mapping from names to types as in the named representation since free variables are distinguished by names.

The notion of free and bound variables depends on the context in which the term that contains the variables is interpreted. For example, the term $(\lambda. x(\lambda. 1\, y))$ contains a bound variable represented as 1, but if we focus on the subterm $(\lambda. 1\, y)$, the variable should be considered free. Therefore, bound variables have to be converted to free variables depending on the context, and vice versa. The term *opening* operation $t[n \mapsto x]$, which converts bound variable n to free variable x, is defined as

$$\begin{array}{rll} n[n \mapsto x] & \coloneqq x & \\ m[n \mapsto x] & \coloneqq m & m \neq n \\ \text{unit}[n \mapsto x] & \coloneqq \text{unit} & \\ (\lambda. t)[n \mapsto x] & \coloneqq \lambda. t[n+1 \mapsto x] & \\ (t_f\, t_a)[n \mapsto x] & \coloneqq t_f[n \mapsto x]\, t_a[n \mapsto x] & \end{array}$$

and $t[0 \mapsto x]$ is abbreviated as t^x. The definition of the opposite *closing* operation, which converts a free variable to a bound variable, is omitted in this thesis because it is not needed to prove type preservation property.

The typing rules of the source language in the locally nameless representation are defined using the opening operation (Figure 5). Note that the rules other than the abs rule are identical to those in the named representation. In the abs rule, the lambda abstraction term $(\lambda. t)$ is well-typed if the subterm t is well-typed. The subterm is opened with some variable x when it is typed. The variable name can be arbitrarily chosen as long as the subterm is well-typed because the semantics of the term does not depend on the choice of the name. Typically, the variable name is chosen from the set of fresh variables that are not in the domain of the typing context. Moreover, it is known that, if the subterm is well-typed when opened with *some* variable name, there are infinitely many names such that the subterm is well-typed when opened with any of the names. Therefore, the typing prerequisite $\Gamma, x{:}T_A \vdash t^x : T_R$ is prefixed with $\forall x \notin l$ stating that the prerequisite should hold for all (infinite many) names that are not in the finite set of names with which the prerequisite does not hold. The universal quantification allows more flexible application of induction hypothesis in proofs [2].

The two-sorted target language is redefined using the locally nameless representation as well (Figure 6). Note that free variables as well as bound variables have two forms: one without a prime for variables from the source term and the other with a prime for variables introduced in transformation. The opening operation is also redefined for each of the two variable sorts. The typing rules are analogous to those in the previous representations.

The term transformation algorithm is the same as in the two-sorted de Bruijn index representation except that the transformation of the free variable has been added:

$$\begin{aligned}
[\![n]\!] &\coloneqq \lambda'.\, 0'\, n \\
[\![x]\!] &\coloneqq \lambda'.\, 0'\, x \\
[\![\text{unit}]\!] &\coloneqq \lambda'.\, 0'\, \text{unit} \\
[\![\lambda.\, t]\!] &\coloneqq \lambda'.\, 0'(\lambda.\, [\![t]\!]) \\
[\![t_f\, t_a]\!] &\coloneqq \lambda'.\, [\![t_f]\!]\big(\lambda'.\, [\![t_a]\!](\lambda'.\, 1'\, 0'\, 2')\big)
\end{aligned}$$

The transformation of types and contexts is the same as defined previously.

To prove the type preservation property in the two-sorted locally nameless representation, we need some lemmas.

Lemma 7: $[\![t]\!][n' \mapsto x'] = [\![t]\!]$.

The proof is by a straightforward induction on the structure of term t.

Lemma 8: $[\![t]\!]^x = [\![t^x]\!]$

This is a corollary of the generalized lemma $[\![t]\!][n \mapsto x] = [\![t[n \mapsto x]]\!]$, which is shown by a straightforward induction on the structure of term t.

$$\begin{array}{rl}
\text{Term} & t \coloneqq n \mid n' \mid x \mid x' \mid \text{unit} \mid \lambda.\,t \mid \lambda'.\,t \mid t\;t \\
\text{Type} & T \coloneqq \text{Unit} \mid T \to T \mid \bot \\
\text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, x\colon T
\end{array}$$

$$\begin{array}{rll}
n[n \mapsto x] & \coloneqq x & \\
m[n \mapsto x] & \coloneqq m & m \neq n \\
m'[n \mapsto x] & \coloneqq m' & \\
\text{unit}[n \mapsto x] & \coloneqq \text{unit} & \\
(\lambda.\,t)[n \mapsto x] & \coloneqq \lambda.\,t[n+1 \mapsto x] & \\
(\lambda'.\,t)[n \mapsto x] & \coloneqq \lambda'.\,t[n \mapsto x] & \\
(t_f\;t_a)[n \mapsto x] & \coloneqq t_f[n \mapsto x]\;t_a[n \mapsto x] & \\
t^x & \coloneqq t[0 \mapsto x] &
\end{array}$$

$$\begin{array}{rll}
m[n' \mapsto x'] & \coloneqq m & \\
n'[n' \mapsto x'] & \coloneqq x' & \\
m'[n' \mapsto x'] & \coloneqq m & m \neq n \\
\text{unit}[n' \mapsto x'] & \coloneqq \text{unit} & \\
(\lambda.\,t)[n' \mapsto x'] & \coloneqq \lambda.\,t[n' \mapsto x'] & \\
(\lambda'.\,t)[n' \mapsto x'] & \coloneqq \lambda'.\,t[(n+1)' \mapsto x'] & \\
(t_f\;t_a)[n' \mapsto x'] & \coloneqq t_f[n' \mapsto x']\;t_a[n' \mapsto x'] & \\
t^{x'} & \coloneqq t[0' \mapsto x'] &
\end{array}$$

$$\frac{x\colon T \in \Gamma}{\Gamma; \Gamma' \vdash x : T}\,\text{var} \qquad \frac{\forall x \notin l \quad \Gamma, x\colon T_A; \Gamma' \vdash t^x : T_R}{\Gamma; \Gamma' \vdash \lambda.\,t : T_A \to T_R}\,\text{abs}$$

$$\frac{x\colon T \in \Gamma'}{\Gamma; \Gamma' \vdash x' : T}\,\text{var'} \qquad \frac{\forall x \notin l \quad \Gamma; \Gamma', x\colon T_A \vdash t^{x'} : T_R}{\Gamma; \Gamma' \vdash \lambda'.\,t : T_A \to T_R}\,\text{abs'}$$

$$\frac{}{\Gamma; \Gamma' \vdash \text{unit} : \text{Unit}}\,\text{unit} \qquad \frac{\Gamma; \Gamma' \vdash t_f : T_A \to T_R \quad \Gamma; \Gamma' \vdash t_a : T_A}{\Gamma; \Gamma' \vdash t_f\;t_a : T_R}\,\text{app}$$

Figure 6: Target language in the two-sorted locally nameless representation

Theorem 9: (Type preservation) If the typing judgment $\Gamma \vdash t : T$ is derivable in the source language, then $[\![\Gamma]\!]; \Gamma' \vdash [\![t]\!] : \neg\neg[\![T]\!]$ is derivable in the target language.

The proof is by the induction on the source typing derivation, which is mostly analogous to that of Theorem 4:

- In the case of the var rule, where $t = x$ and $x\colon T \in \Gamma$, the goal is to derive

$$[\![\Gamma]\!]; \Gamma' \vdash \lambda'.\,0'\,x : \neg\neg[\![T]\!].$$

This can be derived by the abs' rule if

$$\forall k \notin l \quad [\![\Gamma]\!]; \Gamma', k : \neg[\![T]\!] \vdash (0'\, x)^{k'} : \bot$$

can be derived for some l. We choose the domain of context Γ' for l, changing the goal to

$$\forall k \notin \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k : \neg[\![T]\!] \vdash (0'\, x)^{k'} : \bot.$$

By the definition of the opening operation, this is equivalent to

$$\forall k \notin \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k : \neg[\![T]\!] \vdash k'\, x : \bot.$$

This can be derived by applying the app rule. We then have two subgoals:

$$[\![\Gamma]\!]; \Gamma', k : \neg[\![T]\!] \vdash k' : \neg[\![T]\!]$$

$$[\![\Gamma]\!]; \Gamma', k : \neg[\![T]\!] \vdash x : [\![T]\!]$$

Showing the first subgoal is obvious from the var' rule. The second can be obtained by applying Lemma 3 and the var rule to the condition $x : T \in \Gamma$.

- The proof in the case of the unit rule is analogous to that of the var rule. The unit rule in the target language is used to show the final subgoal.
- In the case of the abs rule, where $t = \lambda. t_1$, $T = T_A \to T_R$ and $\forall x \notin l_1;\ \Gamma, x : T_A \vdash t_1^x : T_R$, the goal is to derive

$$[\![\Gamma]\!]; \Gamma' \vdash \lambda'. 0'(\lambda. [\![t_1]\!]) : \neg T_k$$

where $T_k \coloneqq \neg([\![T_A]\!] \to \neg\neg[\![T_R]\!])$. The proof is halfway analogous to that in the case of the var rule. The subgoal we need to prove in this case is

$$[\![\Gamma]\!]; \Gamma', k : \neg T_k \vdash \lambda. [\![t_1]\!]^{k'} : [\![T_A]\!] \to \neg\neg[\![T_R]\!].$$

This can be derived by the abs rule if

$$\forall x \notin l \quad [\![\Gamma]\!], x : [\![T_A]\!]; \Gamma', k : \neg T_k \vdash \left([\![t_1]\!]^{k'}\right)^x : \neg\neg[\![T_R]\!]$$

can be derived for some l. We choose l_1 for l, changing the goal to

$$\forall x \notin l_1 \quad [\![\Gamma]\!], x : [\![T_A]\!]; \Gamma', k : \neg T_k \vdash \left([\![t_1]\!]^{k'}\right)^x : \neg\neg[\![T_R]\!].$$

By Lemma 7, Lemma 8, and the definition of the context transformation, this is equivalent to

$$\forall x \notin l_1 \quad [\![\Gamma, x : T_A]\!]; \Gamma', k : \neg T_k \vdash [\![t_1^x]\!] : \neg\neg[\![T_R]\!],$$

which can be shown by applying the induction hypothesis to the condition $\forall x \notin l_1;\ \Gamma, x : T_A \vdash t_1^x : T_R$.

- In the case of the app rule, where $t = t_f\, t_a$ and $\Gamma \vdash t_f : T_A \to T$ and $\Gamma \vdash t_a : T_A$, the goal is to derive

  $$[\![\Gamma]\!]; \Gamma' \vdash \lambda'. [\![t_f]\!]\big(\lambda'. [\![t_a]\!](\lambda'. 1'\, 0'\, 2')\big) : \neg\neg[\![T]\!].$$

  By applying the abs' rule as in the other cases, the goal is changed to

  $$\forall k \notin \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash \Big([\![t_f]\!]\big(\lambda'. [\![t_a]\!](\lambda'. 1'\, 0'\, 2')\big)\Big)^{k'} : \bot,$$

  which is, by Lemma 7 and the definition of the opening operation, equivalent to

  $$\forall k \notin \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash [\![t_f]\!]\big(\lambda'. [\![t_a]\!](\lambda'. 1'\, 0'\, k')\big) : \bot.$$

  Two subgoals remain after applying the app rule:

  $$[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash [\![t_f]\!] : \neg\neg[\![T_A \to T]\!]$$

  $$[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!] \vdash \lambda'. [\![t_a]\!](\lambda'. 1'\, 0'\, k') : \neg[\![T_A \to T]\!]$$

  The first subgoal is immediate from the induction hypothesis. For the second, we apply the abs' rule to change the subgoal to

  $$\forall f \notin \{k\} \cup \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!], f{:}\,[\![T_A \to T]\!] \vdash \big([\![t_a]\!](\lambda'. 1'\, 0'\, k')\big)^{f'} : \bot,$$

  which is, by Lemma 7 and the definition of the opening operation, equivalent to

  $$\forall f \notin \{k\} \cup \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!], f{:}\,[\![T_A \to T]\!] \vdash [\![t_a]\!](\lambda'. f'\, 0'\, k') : \bot.$$

  Again, two subgoals remain after applying the app rule:

  $$[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!], f{:}\,[\![T_A \to T]\!] \vdash [\![t_a]\!] : \neg\neg[\![T_A]\!]$$

  $$[\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!], f{:}\,[\![T_A \to T]\!] \vdash \lambda'. f'\, 0'\, k' : \neg[\![T_A]\!]$$

  The first subgoal is immediate from the induction hypothesis. We again apply the abs' rule to change the second subgoal to

  $$\forall a \notin \{k, f\} \cup \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!], f{:}\,[\![T_A \to T]\!], a{:}\,[\![T_A]\!] \vdash (f'\, 0'\, k')^{a'} : \bot,$$

  which is equivalent to

  $$\forall a \notin \{k, f\} \cup \mathrm{dom}(\Gamma') \quad [\![\Gamma]\!]; \Gamma', k{:}\,\neg[\![T]\!], f{:}\,[\![T_A]\!] \to \neg\neg[\![T]\!], a{:}\,[\![T_A]\!] \vdash f'\, a'\, k' : \bot.$$

  This subgoal can be derived by the var' and app rules straightforwardly.

3.3 Comparison of Coq Formalization Methods

The author constructed a Coq proof script of the type preservation property for each of the four formalizations introduced in the previous subsections. The four scripts shared the same

Table 1: Lines of Coq proof scripts for simple type preservation

	Unsorted named	**Two-sorted named**	**Two-sorted de Bruijn index**	**Two-sorted locally nameless**
Definition	194	230	160	251
Name lemmas	57	57	0	12
List manipulation	134	12	18	151
Proof of type preservation	120	83	57	124
Total	**505**	**382**	**235**	**538**

structure on the whole. The language syntax and typing rules were inductively defined and the type preservation was proved by induction on the typing derivation in the source language.

The major difference between the four scripts lies in the number of supporting functions and lemmas used. In the named representations, a set of variable names had to be defined that is used in the language definition and proof. Moreover, a function had to be defined that produces a fresh variable name. That cost about 50 lines of Coq proof script including the proof that the function indeed returns a fresh name (the "Name lemmas" row in Table 1). In the two-sorted locally nameless representation, a set of names had to be defined but such a function was not needed.

In the unsorted named and two-sorted locally nameless representations, the author defined and used proof automation tactics to show properties about ordered lists representing typing contexts, which resulted in longer proof scripts (the "List manipulation" row in Table 1). Another reason for a longer proof script in the unsorted named representation is that the freshness of the name had to be verified each time a name-to-type mapping was added to the typing context. In the two-sorted locally nameless representation, more lemmas had to be proved than in the other representations as shown in the previous subsections and that also made the proof script longer (the "Proof of preservation" row in Table 1).

As shown in Table 1, the two-sorted de Bruijn index representation resulted in the shortest proof script. This should be because the typing context was represented as a list of types rather than a list of name-and-type pairs and comparisons of names were not needed in the proof.

Although the two-sorted locally nameless representation resulted in the longest proof script, that should not be interpreted as evidence that the locally nameless representation is less suitable for Coq formalization. The proofs in the named and de Bruijn index representations were relatively simple because term substitution was irrelevant to the proofs. If term substitution were involved in the CPS transformation algorithm, the proofs in those representations would be much more complicated for the reasons stated in the previous subsections. In the next two sections, we deal with type systems containing dependent function types. Such type systems include term substitution in the typing rules. To keep definitions and proofs as simple as possible in spite of term substitution, we use the two-sorted locally nameless representation in the next sections.

4 Proving Semantics Preservation via Singleton Types

In the previous section, we proved that the CPS transformation preserves simple typing in various formalization methods. However, simple types are not so strong as to show the preservation of the full semantics: even if a transformed term is well-typed, it does not necessarily mean that the term has the same semantics as the original term. To show semantics preservation through type preservation, we need to enhance the type system of the target language.

In this section, we define a new target language and a type system that has mainly three features the previous simple type system does not have: subtyping, singleton types, and CPS types. The new type system is defined by adding CPS types (and the unit and the bottom type) to System $\lambda_{\leq\{\}}$ [3], which already has subtyping and singleton types.

The syntax of the new language and type system is similar to that of System $\lambda_{\leq\{\}}$:

$$\begin{array}{rl}
\text{Sort} & s \coloneqq \mathrm{s} \mid \mathrm{c} \\
\text{Term} & t \coloneqq sn \mid sx \mid \mathrm{unit} \mid \lambda s.\, t \mid t\; t \\
\text{Type} & T \coloneqq \mathrm{Unit} \mid \Pi s{:}\, T.\, T \mid \{t : T\} \mid \mathrm{C}(T) \mid \bot \\
\text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, sx : T
\end{array}$$

The syntax is defined using the two-sorted locally nameless representation so that term substitution can be easily handled in the type system.

A new syntax category "sort" is introduced to distinguish the two namespaces instead of using primes. Sort "s" is for variables that come from the source term and sort "c" for variables introduced in the transformation. The bound variable term sn and the free variable term sx are now prefixed by a sort. The lambda abstraction term $\lambda s. t$ also has a sort to indicate which sort of a variable is bound in the abstraction. Variables in a typing context are each prefixed by a sort as well. We use only one typing context in a typing judgment and the context contains a mapping for variables of the both namespaces. This is because a type mapped from a variable in the context may depend on another variable that possibly has the other sort. For example, in the typing context

$$\mathrm{s}x : \mathrm{Unit}, \mathrm{c}y : \{\mathrm{s}x : \mathrm{Unit}\}$$

variable $\mathrm{c}y$ is mapped to the type $\{\mathrm{s}x : \mathrm{Unit}\}$, which depends on variable $\mathrm{s}x$.

Compared to the simply typed lambda calculus, the definition of types is notably different: the function type $\Pi s{:}T. T$ is defined in the dependent manner and the singleton type $\{t : T\}$ and the CPS type $\mathrm{C}(T)$ are added. The dependent function type $\Pi s{:}T_1. T_2$ is similar to the normal function type $T_1 \to T_2$, but it binds a new variable of sort s that can be used in T_2 to denote the argument of the function. The singleton and the CPS types are detailed in the next subsections.

Now that the function type has been redefined dependently, we use the notation "$\neg T$" as an abbreviation for "$\Pi \mathrm{c}{:}T. \bot$" for the rest of this thesis.

The term transformation algorithm from the locally nameless source language (Section 3.2.2) to the two-sorted locally nameless target language is defined analogously to the one in the previous section:

$$\begin{aligned}
[\![n]\!] &:= \lambda \mathrm{c}.\, \mathrm{c}0\, \mathrm{s}n \\
[\![x]\!] &:= \lambda \mathrm{c}.\, \mathrm{c}0\, \mathrm{s}x \\
[\![\mathrm{unit}]\!] &:= \lambda \mathrm{c}.\, \mathrm{c}0\, \mathrm{unit} \\
[\![\lambda. t]\!] &:= \lambda \mathrm{c}.\, \mathrm{c}0 (\lambda \mathrm{s}. [\![t]\!]) \\
[\![t_f\, t_a]\!] &:= \lambda \mathrm{c}. [\![t_f]\!] \big(\lambda \mathrm{c}. [\![t_a]\!] (\lambda \mathrm{c}.\, \mathrm{c}1\, \mathrm{c}0\, \mathrm{c}2)\big)
\end{aligned}$$

4.1 Encapsulating Semantics into Singleton Types

Singleton types [3, 4, 24, 34] are types that denote the semantics of terms by specifying a term to which terms that have the singleton type are equivalent. We write singleton types using braces. For example, the singleton type $\{\mathrm{unit} : \mathrm{Unit}\}$ is the type of terms that are equivalent to the unit value of the unit type. The typing judgment

$$\vdash (\lambda \mathrm{s}.\, \mathrm{s0})\, \mathrm{unit} : \{\mathrm{unit} : \mathrm{Unit}\}$$

holds because the term $((\lambda \mathrm{s}.\, \mathrm{s0})\, \mathrm{unit})$ is beta-equivalent to the unit value (note that $(\lambda \mathrm{s}.\, \mathrm{s0})$ is the identity function). The term specified in a singleton type need not to be a value. The singleton type $\{(\lambda \mathrm{s}.\, \mathrm{s0})\, \mathrm{unit} : \mathrm{Unit}\}$ is a valid type and the typing judgment

$$\vdash \mathrm{unit} : \{(\lambda \mathrm{s}.\, \mathrm{s0})\, \mathrm{unit} : \mathrm{Unit}\}$$

holds as well.

Singleton types can also be used to define the behavior of a function. For example, the identify function $(\lambda \mathrm{s}.\, \mathrm{s0})$ for the unit type has the type $\Pi \mathrm{s{:}Unit}.\{\mathrm{s0} : \mathrm{Unit}\}$, which is more specific than the simple type $\mathrm{Unit} \to \mathrm{Unit}$. The singleton type $\{\mathrm{s0} : \mathrm{Unit}\}$ is used as the return type of the function to denote that the return value has the unit type and is equivalent to the argument value. Note that variable s0, bound by "Πs:", denotes the function argument.

The statement of the type preservation property can be enhanced with singleton types. We use a singleton type for the argument type of a continuation to indicate what value the continuation will receive. For example, the transformation of the unit value is a function that receives a continuation and passes the unit value to the continuation. This can be indicated in the typing judgment using a singleton type:

$$\vdash [\![\mathrm{unit}]\!] : \neg\neg\{\mathrm{unit} : \mathrm{Unit}\}$$

The typing ensures the continuation will receive the unit value. A similar typing is possible for the transformation of a variable: if $\vdash x : T$ holds in the source language, then $\vdash [\![x]\!] : \neg\neg\{sx : [\![T]\!]\}$ holds in the target language. These observations might lead one to conjecture that, if $\vdash t : T$ holds in the source language, then $\vdash [\![t]\!] : \neg\neg\{t : [\![T]\!]\}$ holds in the target language, but it is not true. Subterm t' of the lambda abstraction term $\lambda.\, t'$ has been recursively transformed when the term is passed to a continuation. In such a case, if $\vdash \lambda. t' : T$ holds in the source language, the correct typing in the target language is $\vdash [\![\lambda. t']\!] : \neg\neg\{\lambda \mathrm{s}. [\![t']\!] : [\![T]\!]\}$. Generally, we can obtain the value that will be passed to the continuation by passing the identity function as the continuation to the transformed term, that is, any continuation will be passed a value equivalent to $[\![t]\!](\lambda \mathrm{c}. \mathrm{c}0)$ when applied to the term $[\![t]\!]$. Thus, if $\vdash t : T$ holds in the source language, then $\vdash [\![t]\!] : \neg\neg\{[\![t]\!](\lambda \mathrm{c}. \mathrm{c}0) : [\![T]\!]\}$ holds in the target language, ensuring that any continuation will receive the same value. We will examine the proof of this type preservation property in Section 4.4.

For brevity, the identify function $(\lambda \mathrm{c}. \mathrm{c}0)$ is written as “id” in the rest of this thesis.

4.2 CPS Types

Recall the typing judgment in the previous section:

$$\vdash [\![t]\!] \ : \neg\neg\{[\![t]\!]\,\mathrm{id} : [\![T]\!]\}$$

In this typing, the term being typed appears in the type. We use a *CPS type* to write the type without writing such a term in the type. The CPS type $\mathrm{C}(T)$ denotes a CPS-transformed term that passes a value of type T to the continuation. Using a CPS type, the typing judgment above can be rewritten as

$$\vdash [\![t]\!] : \mathrm{C}([\![T]\!]).$$

The motivation to add CPS types to the type system is that we need them to define the type transformation algorithm. The return type of a function type is transformed into a function type that receives a continuation. The resultant function type needs to be expressed using a CPS type to denote the value the continuation will receive. Now the type transformation is redefined using a CPS type:

$$\begin{aligned} [\![\mathrm{Unit}]\!] &\coloneqq \mathrm{Unit} \\ [\![T_A \rightarrow T_R]\!] &\coloneqq \Pi \mathrm{s}\colon [\![T_A]\!].\mathrm{C}([\![T_R]\!]) \end{aligned}$$

4.3 Type System of the New Target Language

The new type system has four types of judgments:

$$\begin{aligned}
sn[sn \mapsto t] &\coloneqq t \\
s_1 n_1[sn \mapsto t] &\coloneqq s_1 n_1 && s_1 n_1 \neq sn \\
s_1 x[sn \mapsto t] &\coloneqq s_1 x \\
\mathrm{unit}[sn \mapsto t] &\coloneqq \mathrm{unit} \\
(\lambda s.\, t_1)[sn \mapsto t] &\coloneqq \lambda s.\, t_1[s(n+1) \mapsto t] \\
(\lambda s_1.\, t_1)[sn \mapsto t] &\coloneqq \lambda s_1.\, t_1[sn \mapsto t] && s_1 \neq s \\
(t_f\, t_a)[sn \mapsto t] &\coloneqq t_f[sn \mapsto t]\, t_a[sn \mapsto t]
\end{aligned}$$

$$\begin{aligned}
\mathrm{Unit}[sn \mapsto t] &\coloneqq \mathrm{Unit} \\
(\Pi s{:}T_A.T_R)[sn \mapsto t] &\coloneqq \Pi s{:}T_A[sn \mapsto t].T_R[s(n+1) \mapsto t] \\
(\Pi s_1{:}T_A.T_R)[sn \mapsto t] &\coloneqq \Pi s_1{:}T_A[sn \mapsto t].T_R[sn \mapsto t] && s_1 \neq s \\
\{t_1 : T\}[sn \mapsto t] &\coloneqq \{t_1[sn \mapsto t] : T[sn \mapsto t]\} \\
(\mathtt{C}(T))[sn \mapsto t] &\coloneqq \mathtt{C}(T[sn \mapsto t]) \\
\bot\,[sn \mapsto t] &\coloneqq \bot
\end{aligned}$$

Figure 7: Opening operation in the target language with singleton types

Well-formedness of context	$\vdash \Gamma$
Well-formedness of type	$\Gamma \vdash T$
Term typing	$\Gamma \vdash t : T$
Subtyping	$\Gamma \vdash T \leq T$

To define the derivation rules for these judgments, the opening operation need to be defined in advance as in Section 3.2.2. The opening operation is defined for types as well as terms because terms may appear in types (Figure 7). The opening operation defined here is more general than the previous definition in that it substitutes a bound variable with a term instead of a free variable. The generalized opening operation corresponds to the term substitution operation in the usual named representation. We abbreviate $X[sn \mapsto sx]$ as X^{sx} where X is a term or a type.

The derivation rules are defined mutually recursively as in Appendix 1, resembling those of System $\lambda_{\leq\{\}}$. Notable differences from System $\lambda_{\leq\{\}}$ are that the syntax has been adapted to two-sorted locally nameless representation and that the rules for CPS types have been added.

The context well-formedness rules ensure that all types in the context are well-formed and that the variable names in the domain of the context are all unique. The uniqueness of names is useful in rejecting a problematic typing such as

$$u{:}\mathrm{Unit}, u{:}\{\lambda \mathrm{c}.\, u : \Pi \mathrm{c}{:}\mathrm{Unit}.\mathrm{Unit}\} \vdash u : \{\lambda \mathrm{c}.\, u : \Pi \mathrm{c}{:}\mathrm{Unit}.\mathrm{Unit}\}$$

where variable u has either the unit type or the singleton type $\{\lambda c.\, u : \Pi c{:}\text{Unit}.\text{Unit}\}$ depending on the context in which it appears. The rules of type well-formedness, term typing, and subtyping are analogous to those of System $\lambda_{\leq\{\}}$ except for the new rules about the CPS type. The well-formedness of a CPS type is defined straightforwardly.

$$\frac{\Gamma \vdash T}{\Gamma \vdash \mathrm{C}(T)}\ T\text{-cps}$$

The ≤-cpsintro rule converts a singleton type of a specific form to a CPS type.

$$\frac{\Gamma \vdash t : \neg\neg\{t\,\mathrm{id} : T\}}{\Gamma \vdash \{t : \neg\neg\{t\,\mathrm{id} : T\}\} \leq \mathrm{C}(T)}\ \leq\text{-cpsintro}$$

The ≤-cpselim and ≤-cpselimmin rules convert a CPS type to a normal type.

$$\frac{\Gamma \vdash t : \mathrm{C}(T)}{\Gamma \vdash \{t : \mathrm{C}(T)\} \leq \neg\neg\{t\,\mathrm{id} : T\}}\ \leq\text{-cpselim} \qquad \frac{\Gamma \vdash \mathrm{C}(T)}{\Gamma \vdash \mathrm{C}(T) \leq \neg\neg T}\ \leq\text{-cpselimmin}$$

No rules have been added for term typing with a CPS type because a term can have a CPS type from the subtyping with the CPS type.

Lemma 10: (t-cpsintro) If $\Gamma \vdash t : \neg\neg\{t\,\mathrm{id} : T\}$, then $\Gamma \vdash t : \mathrm{C}(T)$.

Proof:

$$\frac{\dfrac{\Gamma \vdash t : \neg\neg\{t\,\mathrm{id} : T\}}{\Gamma \vdash \{t : \neg\neg\{t\,\mathrm{id} : T\}\} \leq \mathrm{C}(T)}\ \leq\text{-cpsintro} \qquad \dfrac{\Gamma \vdash t : \neg\neg\{t\,\mathrm{id} : T\}}{\Gamma \vdash t : \{t : \neg\neg\{t\,\mathrm{id} : T\}\}}\ t\text{-singrefl}}{\Gamma \vdash t : \mathrm{C}(T)}\ t\text{-sub}$$

Lemma 11: (t-cpselim) If $\Gamma \vdash t : \mathrm{C}(T)$, then $\Gamma \vdash t : \neg\neg\{t\,\mathrm{id} : T\}$.

Proof:

$$\frac{\dfrac{\Gamma \vdash t : \mathrm{C}(T)}{\Gamma \vdash \{t : \mathrm{C}(T)\} \leq \neg\neg\{t\,\mathrm{id} : T\}}\ \leq\text{-cpselim} \qquad \dfrac{\Gamma \vdash t : \mathrm{C}(T)}{\Gamma \vdash t : \{t : \mathrm{C}(T)\}}\ t\text{-singrefl}}{\Gamma \vdash t : \neg\neg\{t\,\mathrm{id} : T\}}\ t\text{-sub}$$

An inversion lemma about well-formedness is yet to be proved.

Conjecture 12: (Inversion)

1. If $\Gamma \vdash T$, then $\vdash \Gamma$.
2. If $\Gamma \vdash t : T$, then $\Gamma \vdash T$.
3. If $\Gamma \vdash T_1 \leq T_2$, then $\Gamma \vdash T_1$ and $\Gamma \vdash T_2$.

Lemma 13: (Inversion on T-sing) If $\Gamma \vdash \{t : T\}$, then $\Gamma \vdash t : T$.

Proof: Obvious.

Some typing rules about singleton types are defined as lemmas.

Lemma 14: (t-singsym) If $\Gamma \vdash t_1 : \{t_2 : T\}$, then $\Gamma \vdash t_2 : \{t_1 : T\}$.

Proof:

$$\dfrac{\dfrac{\Gamma \vdash t_2 : T}{\Gamma \vdash t_2 : \{t_2 : T\}}\,t\text{-singrefl} \qquad \dfrac{\Gamma \vdash t_1 : \{t_2 : T\} \quad \dfrac{\Gamma \vdash T}{\Gamma \vdash T \le T}\,\le\text{-refl}}{\Gamma \vdash \{t_2 : T\} \le \{t_1 : T\}}\,\le\text{-singsym}}{\Gamma \vdash t_2 : \{t_1 : T\}}\,t\text{-sub}$$

where $\Gamma \vdash t_2 : T$ and $\Gamma \vdash T$ are from Conjecture 12 and Lemma 13.

Lemma 15: (t-singtrans) If $\Gamma \vdash t_1 : \{t_2 : T\}$ and $\Gamma \vdash t_2 : \{t_3 : T\}$, then $\Gamma \vdash t_1 : \{t_3 : T\}$.

Proof:

$$\dfrac{\Gamma \vdash t_1 : \{t_2 : T\} \qquad \dfrac{\dfrac{\Gamma \vdash t_2 : \{t_3 : T\}}{\Gamma \vdash t_3 : \{t_2 : T\}}\,t\text{-singsym} \quad \dfrac{\Gamma \vdash T}{\Gamma \vdash T \le T}\,\le\text{-refl}}{\Gamma \vdash \{t_2 : T\} \le \{t_3 : T\}}\,\le\text{-singsym}}{\Gamma \vdash t_1 : \{t_3 : T\}}\,t\text{-sub}$$

where $\Gamma \vdash T$ is from Conjecture 12 and Lemma 13.

Lemma 16: (t-singbeta) If $\forall x \notin l,\ \Gamma, sx{:}T_A \vdash t_r^{sx} : T_R^{sx}$ and $\Gamma \vdash t_a : T_A$, then $\Gamma \vdash (\lambda s. t_r)\, t_a :$ $\{t_r[s0 \mapsto t_a] : T_R[s0 \mapsto t_a]\}$.

Proof:

$$\dfrac{\dfrac{\forall x \notin l \quad \dfrac{\Gamma, sx{:}T_A \vdash t_r^{sx} : T_R^{sx}}{\Gamma, sx{:}T_A \vdash t_r^{sx} : \{t_r : T_R\}^{sx}}\,t\text{-singrefl}}{\Gamma \vdash \lambda s. t_r : \Pi s{:}T_A. \{t_r : T_R\}}\,t\text{-abs} \qquad \Gamma \vdash t_a : T_A}{\Gamma \vdash (\lambda s. t_r)\, t_a : \{t_r : T_R\}[s0 \mapsto t_a]}\,t\text{-app}$$

Note that $\{t : T\}^{sx} = \{t^{sx} : T^{sx}\}$ by definition.

The soundness of the type system is a corollary of the following two conjectures, which are yet to be proved. (The reduction rules are defined in Appendix 1.)

Conjecture 17: (Progress) If $\Gamma \vdash t : T$, then either

1. term t is the unit value,
2. term t is a lambda abstraction term, or

3. there exists such term t' that $t \rightarrow_\beta t'$.

Conjecture 18: (Subject reduction)

1. If $\Gamma \vdash t : T$ and $t \rightarrow_\beta t'$, then $\Gamma \vdash t' : T$.
2. If $\Gamma \vdash T_1 \leq T_2$ and $T_1 \rightarrow_\beta T_1'$, then $\Gamma \vdash T_1' \leq T_2$.
3. If $\Gamma \vdash T_1 \leq T_2$ and $T_2 \rightarrow_\beta T_2'$, then $\Gamma \vdash T_1 \leq T_2'$.

4.4 Type Preservation in the CPS Transformation

Lemmas proved in Section 3.2.2 also hold for the new type system with singleton types.

Lemma 19:

1. $[\![t]\!][sn \mapsto sx] = [\![t[n \mapsto x]]\!]$
2. $[\![t]\!][cn \mapsto t'] = [\![t]\!]$
3. $[\![T]\!][cn \mapsto t'] = [\![T]\!]$

Proof: By straightforward induction.

We postulate that bottom types denoting the result values that are returned by continuations can be instantiated with concrete types:

Axiom 20: If $\Gamma \vdash T_1$ and $\Gamma \vdash T_2$, then $\Gamma \vdash \neg\neg T_1 \leq \Pi \mathrm{c}{:}\, (\Pi \mathrm{c}{:}\, T_1. T_2). T_2$.

This axiom is used in the proof of the type preservation property. If the type system of the target language allowed parametric polymorphism, the axiom would not be needed.

Because only one typing context containing the both sorts of names is used in a typing judgment, we need an auxiliary predicate as in Section 3.1.1.

$$\mathrm{CC}(\Gamma; \Gamma') \coloneqq \forall x, \forall T, x{:}\, T \in \Gamma \Rightarrow sx{:}\, [\![T]\!] \in \Gamma'.$$

Using this predicate, the type preservation property is stated as follows.

Conjecture 21: (Type preservation) The typing $\Gamma' \vdash [\![t]\!] : \mathrm{C}([\![T]\!])$ is derivable in the target language if all of the following conditions hold:

- The typing judgment $\Gamma \vdash t : T$ is derivable in the source language where variable names in context Γ are all unique.
- $\mathrm{CC}(\Gamma; \Gamma')$ holds, that is, if $x{:}\, T \in \Gamma$, then $sx{:}\, [\![T]\!] \in \Gamma'$.
- Context Γ' is well-formed, that is, $\vdash \Gamma'$ is derivable in the target language.

The author has not completed the proof of this property. The proof is more complicated than those for the previous target languages are because we need to show term equivalence using singleton types. Here we sketch the point of the proof. The proof is by induction on the source typing derivation $\Gamma \vdash t : T$.

In the case the source typing is derived by the var rule, we need to derive

$$\Gamma' \vdash \lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x : \mathrm{C}(\llbracket T \rrbracket)$$

on the assumption $x{:}\,T \in \Gamma$. The derivation advances as

$$\cfrac{\cfrac{\forall k \notin \mathrm{dom_c}(\Gamma') \quad \cfrac{\dots \vdash \mathrm{c}k : \dots \qquad \dots \vdash \mathrm{s}x : \dots}{\Gamma',\mathrm{c}k{:}\, \neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\} \vdash \mathrm{c}k\, \mathrm{s}x : \bot}\, t\text{-app}}{\Gamma' \vdash \lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x : \neg\neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\}}\, t\text{-abs}}{\Gamma' \vdash \lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x : \mathrm{C}(\llbracket T \rrbracket)}\, t\text{-cpsintro}$$

where $\mathrm{dom}_s(\Gamma) \coloneqq \{x | \mathrm{s}x \in \Gamma\}$. The remaining subgoals are:

1. $\Gamma',\mathrm{c}k{:}\, \neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\} \vdash \mathrm{c}k : \neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\}$
2. $\Gamma',\mathrm{c}k{:}\, \neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\} \vdash \mathrm{s}x : \{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\}$

For the first subgoal, we apply the *t*-var rule, which leaves another subgoal

$$\vdash \Gamma',\mathrm{c}k{:}\, \neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\}.$$

By applying applicable derivation rules repeatedly, we eventually reach the subgoal $\mathrm{s}x \in \Gamma'$, which is sufficed by the assumptions $\mathrm{CC}(\Gamma; \Gamma')$ and $x{:}\,T \in \Gamma$. For the second subgoal, we cannot directly apply the *t*-var rule. We proceed the derivation as

$$\cfrac{\cfrac{\Gamma'' \vdash (\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \{\mathrm{id}\, \mathrm{s}x : \llbracket T \rrbracket\} \quad \Gamma'' \vdash \mathrm{id}\, \mathrm{s}x : \{\mathrm{s}x : \llbracket T \rrbracket\}}{\Gamma'' \vdash (\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \{\mathrm{s}x : \llbracket T \rrbracket\}}\, t\text{-singtrans}}{\Gamma'' \vdash \mathrm{s}x : \{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\}}\, t\text{-singsym}$$

where $\Gamma'' \coloneqq \Gamma',\mathrm{c}k{:}\, \neg\{(\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \llbracket T \rrbracket\}$. After applying Lemma 19, two subgoals remain:

$$\Gamma'' \vdash (\lambda \mathrm{c}.\, \mathrm{c0}\, \mathrm{s}x)\, \mathrm{id} : \{\mathrm{id}\, \mathrm{s}x : \llbracket T \rrbracket[\mathrm{c0} \mapsto \mathrm{id}]\}$$

$$\Gamma'' \vdash \mathrm{id}\, \mathrm{s}x : \{\mathrm{s}x : \llbracket T \rrbracket[\mathrm{c0} \mapsto \mathrm{s}x]\}$$

We apply the *t*-singbeta rule for each of the two subgoals, which yields another four subgoals in total:

$$\begin{array}{ll} \forall k \notin \mathrm{dom_c}(\Gamma'') & \Gamma'',\mathrm{c}k{:}\,\Pi\mathrm{c}{:}\,[\![T]\!].\,[\![T]\!] \vdash \mathrm{c}k\,sx : [\![T]\!]^{\mathrm{c}k} \\ & \Gamma'' \vdash \mathrm{id} : \Pi\mathrm{c}{:}\,[\![T]\!].\,[\![T]\!] \\ \forall v \notin \mathrm{dom_c}(\Gamma'') & \Gamma'',\mathrm{c}v{:}\,[\![T]\!] \vdash \mathrm{c}v : [\![T]\!]^{\mathrm{c}v} \\ & \Gamma'' \vdash sx : [\![T]\!] \end{array}$$

The derivations of these subgoals are straightforward.

The cases of the unit and abs rules are similar to the above. The case of the app rule is much more complex because the transformed term contains more lambda abstractions and the type well-formedness has to be shown each time a name-and-type pair is added to the typing context to type a lambda abstraction. Moreover, the well-formedness of a singleton type requires typing of the term in the singleton type.

The goal we need to derive in the app rule case is

$$\Gamma' \vdash \lambda\mathrm{c}.\,[\![t_f]\!](\lambda\mathrm{c}.\,[\![t_a]\!](\lambda\mathrm{c}.\,\mathrm{c1\ c0\ c2})) : \mathrm{C}([\![T_R]\!])$$

on the assumptions $\Gamma \vdash t_f : T_A \to T_R$ and $\Gamma \vdash t_a : T_A$. In the course of the derivation, it is required to type the innermost subterm:

$$\Gamma',\mathrm{c}k{:}\,T_k,\mathrm{c}f{:}\,T_f,\mathrm{c}a{:}\,T_a \vdash \mathrm{c}f\ \mathrm{c}a\ \mathrm{c}k : \bot$$

where

$$T_k \coloneqq \neg\{(\lambda\mathrm{c}.\,[\![t_f]\!](\lambda\mathrm{c}.\,[\![t_a]\!](\lambda\mathrm{c}.\,\mathrm{c1\ c0\ c2})))\ \mathrm{id} : [\![T_R]\!]\}$$

$$T_f \coloneqq \{[\![t_f]\!]\ \mathrm{id} : [\![T_A \to T_R]\!]\}$$

$$T_a \coloneqq \{[\![t_a]\!]\ \mathrm{id} : [\![T_R]\!]\}$$

We can show that the term $\mathrm{c}f\ \mathrm{c}a$ has the type $\neg\neg\{\mathrm{c}f\ \mathrm{c}a\ \mathrm{id} : [\![T_R]\!]\}$ by straightforward derivation. Then, variable $\mathrm{c}k$ has to have the type $\neg\{\mathrm{c}f\ \mathrm{c}a\ \mathrm{id} : [\![T_R]\!]\}$, which means that the type $\neg\{\mathrm{c}f\ \mathrm{c}a\ \mathrm{id} : [\![T_R]\!]\}$ has to be a supertype of T_k. Applying applicable rules to

$$\ldots \vdash T_k \leq \neg\{\mathrm{c}f\ \mathrm{c}a\ \mathrm{id} : [\![T_R]\!]\}$$

leads to the subgoal

$$\ldots \vdash (\lambda\mathrm{c}.\,[\![t_f]\!](\lambda\mathrm{c}.\,[\![t_a]\!](\lambda\mathrm{c}.\,\mathrm{c1\ c0\ c2})))\ \mathrm{id} : \{\mathrm{c}f\ \mathrm{c}a\ \mathrm{id} : [\![T_R]\!]\}.$$

We divide the subgoal into two using the t-singtrans rule:

$$\ldots \vdash (\lambda\mathrm{c}.\,[\![t_f]\!](\lambda\mathrm{c}.\,[\![t_a]\!](\lambda\mathrm{c}.\,\mathrm{c1\ c0\ c2})))\ \mathrm{id} : \{[\![t_f]\!](\lambda\mathrm{c}.\,[\![t_a]\!](\lambda\mathrm{c}.\,\mathrm{c1\ c0\ id})) : [\![T_R]\!]\}$$

$$\ldots \vdash [\![t_f]\!](\lambda\mathrm{c}.\,[\![t_a]\!](\lambda\mathrm{c}.\,\mathrm{c1\ c0\ id})) : \{\mathrm{c}f\ \mathrm{c}a\ \mathrm{id} : [\![T_R]\!]\}$$

We then continue the proof by applying applicable rules to the subgoals.

5 Preservation of Dependent Types

We used the simply typed lambda calculus as the source language in the previous sections. In this section, we consider the case where the source language is typed with dependent types.

A dependent type is a type that contains (or depends on) a term [5, 17, 37]. For example, consider a typed lambda calculus with primitive natural numbers and lists.

$$\vdash \mathrm{zero} : \mathrm{Nat}$$
$$\vdash \mathrm{succ} : \mathrm{Nat} \to \mathrm{Nat}$$

In the simply typed lambda calculus, all lists have the same type List.

$$\vdash \mathrm{nil} : \mathrm{List}$$
$$\vdash \mathrm{cons} : \mathrm{Nat} \to \mathrm{List} \to \mathrm{List}$$
$$\vdash \mathrm{cons\ zero\ nil} : \mathrm{List}$$

In a dependently typed lambda calculus, the type of a list can contain a number that denotes the length of the list. The type of lists is of the form $\mathrm{List}\, n$ where n denotes the list length.

$$\vdash \mathrm{nil} : \mathrm{List\ zero}$$
$$\vdash \mathrm{cons} : \Pi n{:}\mathrm{Nat}.\ \mathrm{Nat} \to \mathrm{List}\, n \to \mathrm{List}\ (\mathrm{succ}\, n)$$
$$\vdash \mathrm{cons\ zero\ zero\ nil} : \mathrm{List\ (succ\ zero)}$$

The cons function has the dependent function type $\Pi n{:}\mathrm{Nat}.\ \mathrm{Nat} \to \mathrm{List}\, n \to \mathrm{List}\ (\mathrm{succ}\, n)$. When applied to natural number n, the cons function returns another function of the type $\mathrm{Nat} \to \mathrm{List}\, n \to \mathrm{List}\ (\mathrm{succ}\, n)$. For example, the term (cons zero) has the type $\mathrm{Nat} \to \mathrm{List\ zero} \to \mathrm{List\ (succ\ zero)}$. This is justified by the typing rule for application.

$$\frac{\Gamma \vdash t_f : \Pi x{:}T_A.T_R \quad \Gamma \vdash t_a : T_A}{\Gamma \vdash t_f\, t_a : T_R[x \mapsto t_a]}$$

This rule differs from that of the simply typed lambda calculus in that a bound variable is substituted with the argument term in the return type.

The definition of the dependently typed source language is in Appendix 2. The source language uses the locally nameless representation as in the previous section. The definition is based on the standard definition of λLF [5].

5.1 Call-by-Name vs. Call-by-Value

The CPS transformation introduced in Section 2 is the call-by-value version: in the evaluation of a transformed term, terms that correspond to function arguments in the source term are evaluated before they are passed to functions. In the call-by-name version of CPS transformation, however, function arguments are directly passed to functions without evaluation:

$$[\![t_f\ t_a]\!] \coloneqq \lambda k.\, [\![t_f]\!](\lambda v_f.\, v_f [\![t_a]\!] k)$$

Although the difference between the two versions of transformation is very little, it has a great impact on the proof of the type preservation property.

For the call-by-name version of CPS transformation, the type preservation property can be proved by the usual induction on the source typing derivation. Assume we are proving the type preservation property with the target language defined as an extension of the dependently typed source language (Appendix 2) with the bottom type in the two-sorted locally nameless representation. In the case where the last derivation of the source typing is by the t-app rule

$$\frac{\Delta; \Gamma \vdash t_f : \Pi{:}\, T_A.\, T_R \qquad \Delta; \Gamma \vdash t_a : T_A}{\Delta; \Gamma \vdash t_f\ t_a : T_R[0 \mapsto t_a]}\ t\text{-app}$$

we need to derive the following judgment in the target language (for some Δ' and Γ' appropriately corresponding to Δ and Γ).

$$\Delta', \Gamma' \vdash \lambda'.\, [\![t_f]\!](\lambda'.\, 0'[\![t_a]\!]1') : \neg\neg[\![T_R[0 \mapsto t_a]]\!]$$

We carry the proof forward by applying applicable typing rules to derive this judgment. In the course of the derivation, we need to derive the following judgment to type the innermost subterm $0'[\![t_a]\!]1'$.

$$\Delta'; \Gamma', k' {:}\, \neg[\![T_R[0 \mapsto t_a]]\!], f' {:}\, (\Pi{:}\, \neg\neg[\![T_A]\!].\, \neg\neg[\![T_R]\!]) \vdash f'[\![t_a]\!]k' : \bot$$

This can be derived as

$$\frac{\dfrac{\vdash f' : \Pi{:}\, \neg\neg[\![T_A]\!].\, \neg\neg[\![T_R]\!] \qquad \vdash [\![t_a]\!] : \neg\neg[\![T_A]\!]}{\vdash f'[\![t_a]\!] : \neg\neg[\![T_R]\!][0 \mapsto [\![t_a]\!]]} \qquad \vdash k' : \neg[\![T_R[0 \mapsto t_a]]\!]}{\vdash f'[\![t_a]\!]k' : \bot[0 \mapsto k']}$$

where contexts are omitted for brevity. The type $[\![T_R]\!][0 \mapsto [\![t_a]\!]]$ is equal to $[\![T_R[0 \mapsto t_a]]\!]$ (and $\bot[0 \mapsto k]$ is equal to $\bot$) if the opening operation and the type transformation are properly defined. Thus, the dependent type preservation of the call-by-name CPS transformation can be proved by straightforward induction (though the proof is more complicated than in the simply typed case).

For the call-by-value version of CPS transformation, on the other hand, the dependent type preservation cannot be proved in the same way. In the t-app rule case, we need to derive

$$\Delta', \Gamma' \vdash \lambda'.\, [\![t_f]\!]\big(\lambda'.\, [\![t_a]\!](\lambda'.\, 1'\, 0'\, 2')\big) : \neg\neg[\![T_R[0 \mapsto t_a]]\!],$$

which requires the derivation of

$$\Delta'; \Gamma', k' {:} \neg[\![T_R[0 \mapsto t_a]]\!], f' {:} \Pi{:}\, [\![T_A]\!].\, \neg\neg[\![T_R]\!], a' {:} [\![T_A]\!] \vdash f'\, a'\, k' : \bot$$

to type the innermost subterm. The derivation of this judgment will be as follows.

$$\dfrac{\dfrac{\vdash f' : \Pi{:}\, \neg\neg[\![T_A]\!].\, \neg\neg[\![T_R]\!] \quad \vdash a' : \neg\neg[\![T_A]\!]}{\vdash f'\, a' : \neg\neg[\![T_R]\!][0 \mapsto a']} \qquad \vdash k' : \neg[\![T_R[0 \mapsto t_a]]\!]}{\vdash f'\, a'\, k' : \bot[0 \mapsto k']}$$

To validate this derivation, we need to show that the type $[\![T_R]\!][0 \mapsto a']$ is equal to $[\![T_R[0 \mapsto t_a]]\!]$, which requires showing that variable a' is equivalent to term t_a. To show that, we need to know what value variable a' is bound to, but now we only know that the variable has the type $\neg\neg[\![T_A]\!]$. As a result, we cannot validly derive the required judgment above.

5.2 Using Singleton Types to Convey Term Equivalence

The author conjectures that the equivalence above can be shown using singleton types. As shown in Section 4, a singleton type can be used to denote what value is passed to the continuation that is passed to a transformed term. Suppose that the target language is extended with singleton types and CPS types and that the conclusion of the type preservation property is restated as $\Gamma' \vdash [\![t]\!] : \mathrm{C}([\![T]\!])$. Then, the judgment that is required to type the innermost subterm above becomes

$$\Delta'; \Gamma', k' {:} \neg\big\{[\![t_f\; t_a]\!]\mathrm{id} : [\![T_R[0 \mapsto t_a]]\!]\big\}, f' {:} \big\{[\![t_f]\!]\mathrm{id} : \Pi{:}\, [\![T_A]\!].\mathrm{C}([\![T_R]\!])\big\}, a' {:} \{[\![t_a]\!]\mathrm{id} : [\![T_A]\!]\} \\ \vdash f'\, a'\, k' : \bot.$$

Now we know that variable a' is equivalent to $[\![t_a]\!]\mathrm{id}$ from the typing $a' {:} \{[\![t_a]\!]\mathrm{id} : [\![T_A]\!]\}$ in the

$$
\begin{aligned}
\text{Sort} \quad & s \coloneqq \mathrm{s} \mid \mathrm{c} \\
\text{Term} \quad & t \coloneqq sn \mid sx \mid \lambda s.\, t \mid t\, t \\
\text{Type} \quad & T \coloneqq X \mid \Pi s{:}\, T.\, T \mid T\, t \mid \{t : T\} \mid \mathrm{C}(T) \mid \bot \\
\text{Kind} \quad & K \coloneqq * \mid \Pi{:}\, T.\, K
\end{aligned}
$$

$$
\begin{aligned}
sn[sn \mapsto t] &\coloneqq t \\
s_1 n_1[sn \mapsto t] &\coloneqq s_1 n_1 && s_1 n_1 \neq sn \\
s_1 x[sn \mapsto t] &\coloneqq s_1 x \\
(\lambda s.\, t_1)[sn \mapsto t] &\coloneqq \lambda s.\, t_1[s(n+1) \mapsto t] \\
(\lambda s_1.\, t_1)[sn \mapsto t] &\coloneqq \lambda s_1.\, t_1[sn \mapsto t] && s_1 \neq s \\
(t_f\, t_a)[sn \mapsto t] &\coloneqq t_f[sn \mapsto t]\, t_a[sn \mapsto t] \\
t^{sx} &\coloneqq t[sn \mapsto sx]
\end{aligned}
$$

$$
\begin{aligned}
X[sn \mapsto t] &\coloneqq X \\
(\Pi s{:}\, T_A.\, T_R)[sn \mapsto t] &\coloneqq \Pi s{:}\, T_A[sn \mapsto t].\, T_R[s(n+1) \mapsto t] \\
(\Pi s_1{:}\, T_A.\, T_R)[sn \mapsto t] &\coloneqq \Pi s_1{:}\, T_A[sn \mapsto t].\, T_R[sn \mapsto t] && s_1 \neq s \\
T\, t_1\, [sn \mapsto t] &\coloneqq T[sn \mapsto t]\, t_1[sn \mapsto t] \\
\{t_1 : T\}[sn \mapsto t] &\coloneqq \{t_1[sn \mapsto t] : T[sn \mapsto t]\} \\
(\mathrm{C}(T))[sn \mapsto t] &\coloneqq \mathrm{C}(T[sn \mapsto t]) \\
\bot\, [sn \mapsto t] &\coloneqq \bot \\
T^{sx} &\coloneqq T[sn \mapsto sx]
\end{aligned}
$$

Figure 8: Partial definition of the dependently typed target language

$$
\begin{aligned}
[\![n]\!] &\coloneqq \lambda \mathrm{c}.\, \mathrm{c}0\, sn \\
[\![x]\!] &\coloneqq \lambda \mathrm{c}.\, \mathrm{c}0\, \mathrm{s}x \\
[\![\lambda.\, t]\!] &\coloneqq \lambda \mathrm{c}.\, \mathrm{c}0 (\lambda \mathrm{s}.\, [\![t]\!]) \\
[\![t_f\, t_a]\!] &\coloneqq \lambda \mathrm{c}.\, [\![t_f]\!] \big(\lambda \mathrm{c}.\, [\![t_a]\!] (\lambda \mathrm{c}.\, \mathrm{c}1\, \mathrm{c}0\, \mathrm{c}2)\big)
\end{aligned}
$$

$$
\begin{aligned}
[\![X]\!] &\coloneqq X \\
[\![\Pi{:}\, T_A.\, T_R]\!] &\coloneqq \Pi \mathrm{s}{:}\, [\![T_A]\!].\mathrm{C}([\![T_R]\!]) \\
[\![T\, t]\!] &\coloneqq [\![T]\!][\![t]\!] \\
[\![*]\!] &\coloneqq * \\
[\![\Pi{:}\, T.\, K]\!] &\coloneqq \Pi{:}\, [\![T]\!].\, [\![K]\!]
\end{aligned}
$$

Figure 9: CPS transformation to the dependently typed target language

context, we can show that the type $[\![T_R]\!][0 \mapsto a']$ is equal to $[\![T_R[0 \mapsto t_a]]\!]$.

The author has not yet fully settled the definition of the target language that is suitable for proving the type preservation. Of course, the target language needs to have not only dependent types but also singleton types. To the best of the author's knowledge, no research has been done about type systems that have both dependent and singleton types. Giving a full definition of the target language with such a type system and examining properties of the type system is left for future work.

In the rest of this section, a brief summary is given of how to show the equivalence above. First, the syntax of terms and types in the target language and the opening operation are defined as in Figure 8 and the CPS transformation to the target language as in Figure 9. Then, an axiom is assumed that corresponds to Axiom 20:

Axiom 22: If $\Delta; \Gamma \vdash T_1$ and $\Delta; \Gamma \vdash T_2$, then $\Delta; \Gamma \vdash \neg\neg T_1 \leq \Pi \mathrm{c}{:}\, (\Pi \mathrm{c}{:}\, T_1 . T_2). T_2$.

Next, we show some lemmas assuming that the typing and kinding rules of the target language are defined analogously to those of the source language.

Lemma 23: If $\Delta; \Gamma \vdash [\![t]\!][sn \mapsto [\![t_1]\!]\,\mathrm{id}] : T$ and $\Delta; \Gamma \vdash \lambda \mathrm{c}. 0'([\![t_1]\!]\,\mathrm{id}) \equiv [\![t_1]\!] : \mathrm{C}(T)$, then $\Delta; \Gamma \vdash [\![t]\!][sn \mapsto [\![t_1]\!]\,\mathrm{id}] \equiv [\![t[n \mapsto t_1]]\!] : T$.

The proof is by induction on the derivation of $\Delta; \Gamma \vdash [\![t]\!][sn \mapsto [\![t_1]\!]\,\mathrm{id}] : T$. The other assumption is used in the case when t is a variable term.

Lemma 24: If $\Delta; \Gamma \vdash [\![T]\!][sn \mapsto [\![t_1]\!]\,\mathrm{id}] : K$ and $\Delta; \Gamma \vdash \lambda \mathrm{c}. 0'([\![t_1]\!]\,\mathrm{id}) \equiv [\![t_1]\!] : \mathrm{C}(T')$, then $\Delta; \Gamma \vdash [\![T]\!][sn \mapsto [\![t_1]\!]\,\mathrm{id}] \equiv [\![T[n \mapsto t_1]]\!] : K$.

The proof is by induction on the derivation of $\Delta; \Gamma \vdash [\![T]\!][sn \mapsto [\![t_1]\!]\,\mathrm{id}] : K$. Lemma 23 is used to show the equivalence of terms contained in the type.

Lemma 25: If $\Delta; \Gamma \vdash T[sn \mapsto t_1] : K$ and $\Delta; \Gamma \vdash t_1 \equiv t_2 : T'$, then $\Delta; \Gamma \vdash T[sn \mapsto t_1] \equiv T[sn \mapsto t_2] : K$.

The proof is by induction on the derivation of $\Delta; \Gamma \vdash T[sn \mapsto t_1] : K$.

Lemma 26: If $\Delta; \Gamma \vdash [\![T]\!][\mathrm{s}0 \mapsto \mathrm{c}x] : K$ and $\Delta; \Gamma \vdash \mathrm{c}x \equiv [\![t]\!]\,\mathrm{id} : T'$ and $\Delta; \Gamma \vdash \lambda \mathrm{c}. 0'([\![t]\!]\,\mathrm{id}) \equiv [\![t]\!] : \mathrm{C}(T')$, then $\Delta; \Gamma \vdash [\![T]\!][\mathrm{s}0 \mapsto \mathrm{c}x] \equiv [\![T[0 \mapsto t]]\!] : K$.

The proof is immediate from the lemmas above. This lemma establishes the equivalence required for the derivation in the proof of the type preservation property.

$$\dfrac{\dfrac{\vdash f' : \Pi{:}\, \neg\neg[\![T_A]\!]. \neg\neg[\![T_R]\!] \quad \vdash a' : \neg\neg[\![T_A]\!]}{\vdash f'\,a' : \neg\neg[\![T_R]\!][0 \mapsto a']} \qquad \vdash k' : \neg[\![T_R[0 \mapsto t_a]]\!]}{\vdash f'\,a'\,k' : \bot[0 \mapsto k']}$$

Lemma 26 has three assumptions. The first assumption is sufficed by the typing $\vdash f'\,a' : \neg\neg[\![T_R]\!][0 \mapsto a']$ and the inversion lemma from typing to kinding:

Conjecture 27: (Inversion) If $\Delta; \Gamma \vdash t : T$, then $\vdash \Delta; \Gamma$ and $\Delta; \Gamma \vdash T : *$.

The second assumption is satisfied from the typing $a' : \{[\![t_a]\!]\mathrm{id} : [\![T_A]\!]\}$ in the context as mentioned earlier. For the third assumption to be met, we need a stronger induction hypothesis. The type preservation property is restated again:

Conjecture 28: (Type preservation) If the typing judgment $\Delta; \Gamma \vdash t : T$ is derivable in the source language and $\mathrm{CC}(\Delta; \Delta')$ and $\mathrm{CC}(\Gamma; \Gamma')$ holds, then $\Delta'; \Gamma' \vdash \lambda\mathrm{c}. 0'([\![t]\!]\mathrm{id}) \equiv [\![t]\!] : \mathrm{C}([\![T]\!])$ is derivable in the target language where

$$\mathrm{CC}(\Delta; \Delta') := \forall X, \forall K, X{:}K \in \Delta \Rightarrow X{:}[\![K]\!] \in \Delta'$$

$$\mathrm{CC}(\Gamma; \Gamma') := \forall x, \forall T, x{:}T \in \Gamma \Rightarrow \mathrm{s}x{:}[\![T]\!] \in \Gamma'.$$

The new conclusion of the property states not only that the transformed term $[\![t]\!]$ is well-typed in the target language but also that the term is equivalent to $\lambda\mathrm{c}. 0'([\![t]\!]\mathrm{id})$. The equivalence would suffice the third assumption of Lemma 26 in the inductive proof of the property.

6 Related Work

6.1 Typed Lambda Calculi

The definition of the target language with singleton types introduced in Section 4 is based on Aspinall's simple type system with singleton types and subtyping [3]. In this type system, the notion of term equivalence is integrated with typing with singleton types. Stone and Harper created another type system with singleton types in which the notion of term equivalence is separated from typing [34]. Their type system is more complex than Aspinall's due to the separated notion, but is capable of handling extensional equivalence. The target language in this thesis is based on Aspinall's type system in favor of simplicity. Extensional equivalence is not needed to show the type preservation property of CPS transformation.

6.2 Representations of Lambda Calculi in Coq

The basic idea of the locally nameless representation already appeared in de Bruijn's paper that first introduced the de Bruijn index representation [16]. He used numbers for bound variables while using names for free variables. Leroy demonstrated a practical proof of the soundness of a type system using the locally nameless representation [19]. Aydemir [2] and Charguéraud [9] investigated properties of the locally nameless representation. Charguéraud also showed an approach to proving the semantics preservation of a CPS transformation algorithm in the (unsorted) locally nameless representation. With the unsorted representation, he had to track the correspondence of variables between before and after the transformation and thus had to prove many lemmas.

Formalization methods other than the de Bruijn index and the locally nameless representations include higher-order abstract syntax (HOAS) [22, 28] and nominal logic [30]. Neither HOAS nor nominal logic is known to be fully compatible with the calculus of inductive constructions (CIC), which is Coq's fundamental type system [14]. Chlipala proposed a variation of HOAS that is suitable for Coq formalization [11]. Aydemir et al. presented a set of axioms to allow formalization using nominal logic in Coq [1], but it is not based on the induction principles of CIC. Westbrook et al. proposed an extension of CIC to allow formalization using nominal logic within the type system [36]. Formalizations using the de Bruijn index and the locally nameless representations, on the other hand, are feasible within CIC.

6.3 Verification of CPS Transformation

The idea of using two namespaces in the target language of CPS transformation was coined by Dargaye and Leroy [13]. They used two-sorted de Bruijn index representation to prove semantics preservation of CPS transformation with the Coq proof assistant. Minamide and Okuma proved the semantics preservation of several CPS transformation algorithms using Isabelle/HOL [23]. They used different sets of variable names for the source and target languages to avoid the difficulty about the naming of new variables. Both of these dealt with an untyped language and did not discuss type preservation in CPS transformation.

Barthe et al. showed type preservation property of CPS transformation in typed lambda calculi (including dependently typed lambda calculi) [6]. They emphasized that transformed terms should not contain types so that the term transformation algorithm does not depend on the type transformation. They dealt with the call-by-name version of Plotkin's CPS transformation, but not with the call-by-value version. Shao et al. showed the type preservation of the call-by-value transformation of a dependently typed lambda calculus where terms appearing types are translated into types [32].

6.4 Mechanized Verification of Compilers

The research in this thesis was originally aiming to create a certified compiler that compiles a dependently typed language such as Hoare Type Theory [26, 27] into a dependently typed assembly language such as Certified Assembly Programming [15, 38]. A certified compiler is a compiler that is verified to produce a correct assembly code in the sense that the assembly code preserves the semantics of the source code.

Chlipala created certified compilers that translate typed lambda calculi into typed assembly [10]. He used higher-order abstract syntax to prove the semantics preservation property of the compilers with Coq formalization. Although his compiler preserves typing as well as semantics, he did not use types to denote the semantics of terms.

The CompCert project aims to create a certified and realistic compiler from a subset of C to PowerPC assembly [8, 18, 20]. The CompCert compiler is accompanied by a large amount of Coq proof script that shows its semantics preservation property. Since C is not a type-safe language, the compiler does not ensure the safety of output assembly code.

Neither of Chlipala's compiler and CompCert deals with dependently typed source and target languages.

7 Conclusion

This study investigated type preservation property of CPS transformation with various type systems and formalization methods. We first examined four types of formalization for the simply typed lambda calculus: the unsorted named, two-sorted named, two-sorted de Bruijn index, and two-sorted locally nameless representations. We then compared Coq proof scripts of the type preservation property of the CPS transformation that were written in those formalizations. It was shown that the two-sorted representations make the proofs simpler and that the locally nameless representation is more suitable than the named or de Bruijn index representations when term substitution is involved in the language definition and proofs.

A type system with singleton types and subtyping was defined in the two-sorted locally nameless representation. The semantics of CPS-transformed terms were expressed using singleton types. We saw that it is much more complicated to prove the type preservation property in the CPS transformation from the source language typed with simple types to the target language typed with singleton types than that to the simply typed target language. Although the proof is not yet finished, a sketch of the proof was presented.

It was shown that a proof of dependent type preservation in the call-by-value CPS transformation requires a proof of term equivalence that is not required in the case of the call-by-name CPS transformation. Conveying the semantics of terms with singleton types was proposed to prove the required equivalence.

References

1. Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. “Nominal Reasoning Techniques in Coq.” In *Electron. Notes Theor. Comput. Sci.* Vol. 174, No. 5, 2007, pp. 69–77.
2. Brian Aydemir, et al. “Engineering formal metatheory.” In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL ’08). ACM, New York, 2008, pp. 3–15.
3. David Aspinall. “Subtyping with singleton types.” In *Proceedings of the Computer science logic* (CSL ’94), Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer-Verlag, Berlin, 1995, LNCS, Vol. 933, pp. 1–15.
4. David Aspinall. “Type systems for modular programs and specifications.” Ph. D. thesis, Department of Computer Science, University of Edinburgh, 1997.
5. David Aspinall and Martin Hofmann. “Dependent Types.” In *Advanced topics in types and programming languages*, Benjamin C. Pierce (Ed.). The MIT Press, 2005, pp. 45–86.

6. Gilles Barthe, John Hatcliff, and Morten H.B. Sørensen. "CPS transformations and applications: The cube and beyond." In *Higher-Order and Symbolic Computation*. Springer, Netherlands, 1999, Vol. 12, No. 2, pp. 125–170.
7. Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag, New York, 2004.
8. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal verification of a C compiler front-end." In *Proceedings of the 14th international symposium on Formal methods* (FM '06), Jayadev Misra et al (Eds.). Springer-Verlag, Berlin, 2006, LNCS, Vol. 4085, pp. 460–475.
9. Arthur Charguéraud. "The locally nameless representation." To appear in *Journal of automated reasoning*. 2011.
10. Adam Chlipala. "A certified type-preserving compiler from lambda calculus to assembly language." In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '07). ACM, New York, 2007, pp. 54–65.
11. Adam Chlipala. "Parametric higher-order abstract syntax for mechanized semantics." In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming* (ICFP '08). ACM, New York, 2008, pp. 143–156.
12. The Coq development team. The Coq proof assistant. 1989–2010. http://coq.inria.fr/.
13. Zaynah Dargaye and Xavier Leroy. "Mechanized verification of CPS transformations." In *Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning* (LPAR '07), Nachum Dershowitz and Andrei Voronkov (Eds.). Springer-Verlag, Berlin, 2007, pp. 211–225.
14. Joëlle Despeyroux, Amy Felty, and André Hirschowitz. "Higher-order abstract syntax in Coq." In *Typed lambda calculi and applications*, Mariangiola Dezani-Ciancaglini and Gordon Plotkin (Eds.). Springer-Verlag, Berlin, 1995, LNCS, Vol. 902, pp. 124–138.
15. Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. "Modular verification of assembly code with stack-based control abstractions." In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*

(PLDI ’06). ACM, New York, 2006, pp. 401–414.

16. Nicolaas Govert de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem.” In *Indagationes Mathematicae* (Proceedings). Elsevier, Vol. 75, No. 5, 1972, pp. 381–392.
17. Robert Harper, Furio Honsell, and Gordon Plotkin. “A framework for defining logics.” *J. ACM.* Vol. 40, No. 1, 1993, pp. 143–184.
18. Xavier Leroy. “Formal certification of a compiler back-end or: programming a compiler with a proof assistant.” In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL ’06). ACM, New York, 2006, pp. 42–54.
19. Xavier Leroy. “A locally nameless solution to the POPLmark challenge.” Technical report 6098, INRIA, 2007.
20. Xavier Leroy. “Formal verification of a realistic compiler.” In *Commun. ACM*. 2009, Vol. 52, No. 7, pp. 107–115.
21. Ralph Loader. “Notes on simply typed lambda calculus.” Technical report 381, LFCS, University of Edinburgh, 1998.
22. Dale Miller. “Abstract syntax for variable binders: An overview.” In *Computational Logic — CL 2000*, John Lloyd et al. (Eds.). Springer-Verlag, Berlin, 2000, LNCS, Vol. 1861, pp. 239–253.
23. Yasuhiko Minamide and Koji Okuma. “Verifying CPS transformations in Isabelle/HOL.” In *Proceedings of the 2003 ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (MERLIN ’03). ACM, New York, 2003, pp. 1–8.
24. Stefan Monnier and David Haguenauer. “Singleton types here, singleton types there, singleton types everywhere.” In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification* (PLPV ’10). ACM, New York, 2010, pp. 1–8.
25. Magnus O. Myreen. “Verified just-in-time compiler on x86.” In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL ’10). ACM, New York, 2010, pp. 107–118.

26. Aleksandar Nanevski and Greg Morrisett. “Dependent type theory of stateful higher-order functions.” Technical report TR-24-05, Harvard University, 2005.

27. Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. “Polymorphism and separation in hoare type theory.” In *Proceedings of the 11th ACM SIGPLAN international conference on Functional programming* (ICFP ’06). ACM, New York, 2006, pp. 62–73.

28. F. Pfenning and C. Elliot. “Higher-order abstract syntax.” In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (PLDI ’88), R. L. Wexelblat (Ed.). ACM, New York, 1988, pp. 199–208.

29. Benjamin C. Pierce. *Types and programming languages*. The MIT Press, London, 2002.

30. Andrew M. Pitts. “Nominal logic, a first order theory of names and binding.” In *Inf. Comput.* Vol. 186, No. 2, 2003, pp. 165–193.

31. Gordon Plotkin. “Call-by-name, call-by-value and the λ-calculus.” In *Theoretical Computer Science*, 1975, Vol. 1, No. 2, pp. 125–159.

32. Zhong Shao, Valery Trifonov, Bratin Saha, and Nikolaos Papaspyrou. “A type system for certified binaries.” In *ACM Trans. Program. Lang. Syst.* Vol. 27, No. 1, 2005, pp. 1–45.

33. Christopher A. Stone. “Singleton kinds and singleton types.” Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, 2000.

34. Christopher A. Stone and Robert Harper. “Extensional equivalence and singleton types.” In *ACM Trans. Comput. Logic*. ACM, New York, Vol. 7, No. 4, 2006, pp. 676–722.

35. Gerald Jay Sussman and Guy L. Steele, Jr. “Scheme: An interpreter for extended lambda calculus.” In *Higher-Order and Symbolic Computation*. Springer, Netherlands, 1998, Vol. 11, No. 4, pp. 405–439.

36. Edwin Westbrook, Aaron Stump, and Evan Austin. “The calculus of nominal inductive constructions: an intensional approach to encoding name-bindings.” In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice* (LFMTP ’09). ACM, New York, 2009, pp. 74–83.

37. Hongwei Xi and Frank Pfenning. “Dependent types in practical programming.” In *Proceed-*

ings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99). ACM, New York, 1999, pp. 214–227.

38. Dachuan Yu, Nadeem A. Hamid and Zhong Shao. "Building certified libraries for PCC: dynamic storage allocation." In *Science of Computer Programming* (ESOP 2003). Elsevier, Vol. 50, No. 1–3, 2004, pp. 101–127.

Appendix 1. Definition of the Target Language with Singleton Types

A 1.1 Syntax

$$\begin{array}{rl}
\text{Sort} & s \coloneqq \mathrm{s} \mid \mathrm{c} \\
\text{Term} & t \coloneqq sn \mid sx \mid \mathrm{unit} \mid \lambda s.\, t \mid t\ t \\
\text{Type} & T \coloneqq \mathrm{Unit} \mid \Pi s{:}\, T.\, T \mid \{t : T\} \mid \mathrm{C}(T) \mid \bot \\
\text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, sx : T
\end{array}$$

A 1.2 Opening

$$\begin{array}{rll}
sn[sn \mapsto t] & \coloneqq t & \\
s_1 n_1[sn \mapsto t] & \coloneqq s_1 n_1 & s_1 n_1 \neq sn \\
s_1 x[sn \mapsto t] & \coloneqq s_1 x & \\
\mathrm{unit}[sn \mapsto t] & \coloneqq \mathrm{unit} & \\
(\lambda s.\, t_1)[sn \mapsto t] & \coloneqq \lambda s.\, t_1[s(n+1) \mapsto t] & \\
(\lambda s_1.\, t_1)[sn \mapsto t] & \coloneqq \lambda s_1.\, t_1[sn \mapsto t] & s_1 \neq s \\
\left(t_f\ t_a\right)[sn \mapsto t] & \coloneqq t_f[sn \mapsto t]\ t_a[sn \mapsto t] & \\
t^{sx} & \coloneqq t[sn \mapsto sx] &
\end{array}$$

$$\begin{array}{rll}
\mathrm{Unit}[sn \mapsto t] & \coloneqq \mathrm{Unit} & \\
(\Pi s{:}\, T_A.\, T_R)[sn \mapsto t] & \coloneqq \Pi s{:}\, T_A[sn \mapsto t].\, T_R[s(n+1) \mapsto t] & \\
(\Pi s_1{:}\, T_A.\, T_R)[sn \mapsto t] & \coloneqq \Pi s_1{:}\, T_A[sn \mapsto t].\, T_R[sn \mapsto t] & s_1 \neq s \\
\{t_1 : T\}[sn \mapsto t] & \coloneqq \{t_1[sn \mapsto t] : T[sn \mapsto t]\} & \\
\left(\mathrm{C}(T)\right)[sn \mapsto t] & \coloneqq \mathrm{C}(T[sn \mapsto t]) & \\
\bot\ [sn \mapsto t] & \coloneqq \bot & \\
T^{sx} & \coloneqq T[sn \mapsto sx] &
\end{array}$$

A 1.3 Context Well-formedness

$$\frac{}{\vdash \emptyset}\Gamma\text{-empty} \qquad \frac{sx \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash T}{\vdash \Gamma, sx{:}\, T}\Gamma\text{-cons}$$

A 1.4 Type Well-formedness

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathrm{Unit}}T\text{-unit} \qquad \frac{\forall x \notin l \quad \Gamma, sx{:}\, T_A \vdash T_R^{sx}}{\Gamma \vdash \Pi s{:}\, T_A.\, T_R}T\text{-prod}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{t : T\}}T\text{-sing} \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \mathrm{C}(T)}T\text{-cps}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \bot}T\text{-bot}$$

A 1.5 Term Typing

$$\frac{\vdash \Gamma \quad sx{:}\,T \in \Gamma}{\Gamma \vdash sx : T}t\text{-var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{unit} : \mathrm{Unit}}t\text{-unit}$$

$$\frac{\forall x \notin l \quad \Gamma, sx{:}\,T_A \vdash t^{sx} : T_R^{sx}}{\Gamma \vdash \lambda s.\, t : \Pi s{:}\,T_A.\,T_R}t\text{-abs} \qquad \frac{\Gamma \vdash t_f : \Pi s{:}\,T_A.\,T_R \quad \Gamma \vdash t_a : T_A}{\Gamma \vdash t_f\, t_a : T_R[s0 \mapsto t_a]}t\text{-app}$$

$$\frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash t : T_1}{\Gamma \vdash t : T_2}t\text{-sub} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash t : \{t : T\}}t\text{-singrefl}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_{A2} \leq T_{A1} \\ \forall x \notin l \quad \Gamma, sx{:}\,T_{A2} \vdash t_1^{sx} : \{t_2 : T_{R2}\}^{sx} \\ \forall x \notin l \quad \Gamma, sx{:}\,T_{A1} \vdash t_1^{sx} : T_{R1}^{sx}\end{array}}{\Gamma \vdash \lambda s.\, t_1 : \{\lambda s.\, t_2 : \Pi s{:}\,T_{A2}.\,T_{R2}\}}t\text{-singabs}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_{f1} : \left\{t_{f2} : \Pi s{:}\,T_A.\,T_R\right\} \\ \Gamma \vdash t_{a1} : \{t_{a2} : T_A\}\end{array}}{\Gamma \vdash t_{f1}\, t_{a1} : \left\{t_{f2}\, t_{a2} : T_R[s0 \mapsto t_{a1}]\right\}}t\text{-singapp}$$

A 1.6 Subtyping

$$\frac{\Gamma \vdash T}{\Gamma \vdash T \leq T}\leq\text{-refl} \qquad \frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3}\leq\text{-trans}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_{A2} \leq T_{A1} \\ \forall x \notin l \quad \Gamma, sx{:}\,T_{A2} \vdash T_{R1}^{sx} \leq T_{R2}^{sx} \\ \forall x \notin l \quad \Gamma, sx{:}\,T_{A1} \vdash T_{R1}^{sx}\end{array}}{\Gamma \vdash \Pi s{:}\,T_{A1}.\,T_{R1} \leq \Pi s{:}\,T_{A2}.\,T_{R2}}\leq\text{-prod}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{t : T\} \leq T}\leq\text{-singinv} \qquad \frac{\Gamma \vdash t_1 : \{t_2 : T_1\} \quad \Gamma \vdash T_1 \leq T_2}{\Gamma \vdash \{t_2 : T_1\} \leq \{t_1 : T_2\}}\leq\text{-singsym}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{t : T\} \leq \left\{t : \{t : T\}\right\}}\leq\text{-singiter}$$

$$\frac{\Gamma \vdash t : \neg\neg\{t\, \mathrm{id} : T\}}{\Gamma \vdash \left\{t : \neg\neg\{t\, \mathrm{id} : T\}\right\} \leq \mathrm{C}(T)}\leq\text{-cpsintro}$$

$$\frac{\Gamma \vdash t : \mathrm{C}(T)}{\Gamma \vdash \{t : \mathrm{C}(T)\} \leq \neg\neg\{t\ \mathrm{id} : T\}}\leq\text{-cpselim} \qquad \frac{\Gamma \vdash \mathrm{C}(T)}{\Gamma \vdash \mathrm{C}(T) \leq \neg\neg T}\leq\text{-cpselimmin}$$

A 1.7 Term Reduction

$$\frac{}{(\lambda s. t_r)\ t_a \rightarrow_\beta t_r[s0 \mapsto t_a]}t\beta\text{-beta}$$

$$\frac{t_{f1} \rightarrow_\beta t_{f2}}{t_{f1}\ t_a \rightarrow_\beta t_{f2}\ t_a}t\beta\text{-app1} \qquad \frac{t_{a1} \rightarrow_\beta t_{a2}}{t_f\ t_{a1} \rightarrow_\beta t_f\ t_{a2}}t\beta\text{-app2}$$

A 1.8 Type Reduction

$$\frac{T_{A1} \rightarrow_\beta T_{A2}}{\Pi s{:}\, T_{A1}. T_R \rightarrow_\beta \Pi s{:}\, T_{A2}. T_R}T\beta\text{-prodarg} \qquad \frac{\forall x \notin l \quad T_{R1}^{sx} \rightarrow_\beta T_{R2}^{sx}}{\Pi s{:}\, T_A. T_{R1} \rightarrow_\beta \Pi s{:}\, T_A. T_{R2}}T\beta\text{-prodret}$$

$$\frac{t_1 \rightarrow_\beta t_2}{\{t_1 : T\} \rightarrow_\beta \{t_2 : T\}}T\beta\text{-singterm} \qquad \frac{T_1 \rightarrow_\beta T_2}{\{t : T_1\} \rightarrow_\beta \{t : T_2\}}T\beta\text{-singtype}$$

$$\frac{T_1 \rightarrow_\beta T_2}{\mathrm{C}(T_1) \rightarrow_\beta \mathrm{C}(T_2)}T\beta\text{-cps}$$

Appendix 2. Definition of the Dependently Typed Source Language

A 2.1 Syntax

$$\begin{array}{rl}
\text{Term} & t \coloneqq n \mid x \mid \lambda. t \mid t\ t \\
\text{Type} & T \coloneqq X \mid \Pi{:}\, T. T \mid T\ t \\
\text{Kind} & K \coloneqq * \mid \Pi{:}\, T. K \\
\text{Typing context} & \Gamma \coloneqq \emptyset \mid \Gamma, x{:}\, T \\
\text{Kinding context} & \Delta \coloneqq \emptyset \mid \Delta, X{:}\, K
\end{array}$$

A 2.2 Opening

$$\begin{array}{rll}
n[n \mapsto t] & \coloneqq t & \\
m[n \mapsto t] & \coloneqq m & m \neq n \\
x[n \mapsto t] & \coloneqq x & \\
(\lambda. t_1)[n \mapsto t] & \coloneqq \lambda. t_1[n+1 \mapsto t] & \\
(t_f\ t_a)[n \mapsto t] & \coloneqq t_f[n \mapsto t]\ t_a[n \mapsto t] & \\
t^x & \coloneqq t[0 \mapsto x] &
\end{array}$$

$$\begin{aligned}
X[n \mapsto t] &\coloneqq X \\
(\Pi{:}\, T_A . T_R)[n \mapsto t] &\coloneqq \Pi{:}\, T_A[n \mapsto t] . T_R[n+1 \mapsto t] \\
(T\ t_1)[n \mapsto t] &\coloneqq T[n \mapsto t]\ t_1[n \mapsto t] \\
T^x &\coloneqq T[0 \mapsto x]
\end{aligned}$$

$$\begin{aligned}
*[n \mapsto t] &\coloneqq * \\
(\Pi{:}\, T . K)[n \mapsto t] &\coloneqq \Pi{:}\, T[n \mapsto t] . K[n+1 \mapsto t] \\
K^x &\coloneqq K[0 \mapsto x]
\end{aligned}$$

A 2.3 Context Well-formedness

This judgment is of the form $\vdash \Delta; \Gamma$.

$$\frac{}{\vdash \emptyset; \emptyset}\,\Delta\Gamma\text{-empty} \qquad \frac{X \notin \mathrm{dom}(\Delta) \quad \Delta; \Gamma \vdash K}{\vdash \Delta, X{:}\, K; \Gamma}\,\Delta\text{-cons} \qquad \frac{x \notin \mathrm{dom}(\Gamma) \quad \Delta; \Gamma \vdash T : *}{\vdash \Delta; \Gamma, x{:}\, T}\,\Gamma\text{-cons}$$

A 2.4 Kind Well-formedness

This judgment is of the form $\Delta; \Gamma \vdash K$.

$$\frac{\vdash \Delta; \Gamma}{\Delta; \Gamma \vdash *}\,K\text{-}* \qquad \frac{\forall x \notin l \quad \Delta; \Gamma, x{:}\, T \vdash K^x}{\Delta; \Gamma \vdash \Pi{:}\, T . K}\,K\text{-prod}$$

A 2.5 Type Kinding

This judgment is of the form $\Delta; \Gamma \vdash T{:}\, K$.

$$\frac{\vdash \Delta; \Gamma \quad X{:}\, K \in \Delta}{\Delta; \Gamma \vdash X{:}\, K}\,T\text{-var} \qquad \frac{\forall x \notin l \quad \Delta; \Gamma, x{:}\, T_A \vdash T_R^x : *}{\Delta; \Gamma \vdash \Pi{:}\, T_A . T_R : *}\,T\text{-prod}$$

$$\frac{\Delta; \Gamma \vdash T : \Pi{:}\, T_A . K \quad \Delta; \Gamma \vdash t : T_A}{\Delta; \Gamma \vdash T\ t : K[0 \mapsto t]}\,T\text{-app} \qquad \frac{\Delta; \Gamma \vdash K_1 = K_2 \quad \Delta; \Gamma \vdash T : K_1}{\Delta; \Gamma \vdash T : K_2}\,T\text{-eq}$$

A 2.6 Term Typing

This judgment is of the form $\Delta; \Gamma \vdash t : T$.

$$\frac{\vdash \Delta; \Gamma \quad x{:}\, T \in \Gamma}{\Delta; \Gamma \vdash x : T}\,t\text{-var} \qquad \frac{\forall x \notin l \quad \Delta; \Gamma, x{:}\, T_A \vdash t^x : T_R^x}{\Delta; \Gamma \vdash \lambda . t : \Pi{:}\, T_A . T_R}\,t\text{-abs}$$

$$\frac{\Delta; \Gamma \vdash t_f : \Pi{:}\, T_A . T_R \quad \Delta; \Gamma \vdash t_a : T_A}{\Delta; \Gamma \vdash t_f\ t_a : T_R[0 \mapsto t_a]}\,t\text{-app} \qquad \frac{\Delta; \Gamma \vdash T_1 = T_2 : * \quad \Delta; \Gamma \vdash t : T_1}{\Delta; \Gamma \vdash t : T_2}\,t\text{-eq}$$

A 2.7 Kind Equivalence

This judgment is of the form $\Delta; \Gamma \vdash K \equiv K$.

$$\frac{\Delta; \Gamma \vdash K}{\Delta; \Gamma \vdash K \equiv K} K_{\equiv}\text{-refl} \qquad \frac{\Delta; \Gamma \vdash K_1 \equiv K_2}{\Delta; \Gamma \vdash K_2 \equiv K_1} K_{\equiv}\text{-sym}$$

$$\frac{\Delta; \Gamma \vdash K_1 \equiv K_2 \quad \Delta; \Gamma \vdash K_2 \equiv K_3}{\Delta; \Gamma \vdash K_1 \equiv K_3} K_{\equiv}\text{-trans}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash T_1 \equiv T_2 : * \\ \forall x \notin l \quad \Delta; \Gamma, x\colon T_1 \vdash K_1^x \equiv K_2^x\end{array}}{\Delta; \Gamma \vdash \Pi\colon T_1 . K_1 \equiv \Pi\colon T_2 . K_2} K_{\equiv}\text{-prod}$$

A 2.8 Type Equivalence

This judgment is of the form $\Delta; \Gamma \vdash T \equiv T : K$.

$$\frac{\Delta; \Gamma \vdash T : K}{\Delta; \Gamma \vdash T \equiv T : K} T_{\equiv}\text{-refl} \qquad \frac{\Delta; \Gamma \vdash T_1 \equiv T_2 : K}{\Delta; \Gamma \vdash T_2 \equiv T_1 : K} T_{\equiv}\text{-sym}$$

$$\frac{\Delta; \Gamma \vdash T_1 \equiv T_2 : K \quad \Delta; \Gamma \vdash T_2 \equiv T_3 : K}{\Delta; \Gamma \vdash T_1 \equiv T_3 : K} T_{\equiv}\text{-trans}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash T_{A1} \equiv T_{A2} : * \\ \forall x \notin l \quad \Delta; \Gamma, x\colon T_{A1} \vdash T_{R1}^x \equiv T_{R2}^x : *\end{array}}{\Delta; \Gamma \vdash \Pi\colon T_{A1} . T_{R1} \equiv \Pi\colon T_{A2} . T_{R2} : *} T_{\equiv}\text{-prod}$$

A 2.9 Term Equivalence

This judgment is of the form $\Delta; \Gamma \vdash t \equiv t : T$.

$$\frac{\Delta; \Gamma \vdash t : T}{\Delta; \Gamma \vdash t \equiv t : T} t_{\equiv}\text{-refl} \qquad \frac{\Delta; \Gamma \vdash t_1 \equiv t_2 : T}{\Delta; \Gamma \vdash t_2 \equiv t_1 : T} t_{\equiv}\text{-sym}$$

$$\frac{\Delta; \Gamma \vdash t_1 \equiv t_2 : T \quad \Delta; \Gamma \vdash t_2 \equiv t_3 : T}{\Delta; \Gamma \vdash t_1 \equiv t_3 : T} t_{\equiv}\text{-trans}$$

$$\frac{\forall x \notin l \quad \Delta; \Gamma, x\colon T_A \vdash t_1^x \equiv t_2^x : T_R^x}{\Delta; \Gamma \vdash \lambda . t_1 \equiv \lambda . t_2 : \Pi\colon T_A . T_R} t_{\equiv}\text{-abs} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash t_{f1} \equiv t_{f2} : \Pi\colon T_A . T_R \\ \forall x \notin l \quad \Delta; \Gamma, x\colon T_A \vdash t_{a1} \equiv t_{a2} : T_A\end{array}}{\Delta; \Gamma \vdash t_{f1}\, t_{a1} \equiv t_{f2}\, t_{a2} : T_R[0 \mapsto t_{a1}]} t_{\equiv}\text{-app}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash t_a : T_A \\ \forall x \notin l \quad \Delta; \Gamma, x\colon T_A \vdash t_r^x : T_R^x\end{array}}{\Delta; \Gamma \vdash (\lambda . t_r)\, t_a \equiv t_r[0 \mapsto t_a] : T_R[0 \mapsto t_a]} t_{\equiv}\text{-beta} \qquad \frac{\Delta; \Gamma \vdash t : \Pi\colon T_A . T_R}{\Delta; \Gamma \vdash \lambda . t\, 0 \equiv t : \Pi\colon T_A . T_R} t_{\equiv}\text{-eta}$$