A Mostly CPS, Partly ANF Translation of Dependent Types

Youyou Cong
cong@c.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Hironori Kawazoe
kawazoe.h@prg.is.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Institute of Technology
Tokyo, Japan

Abstract

Type-preserving compilation is an approach to building reliable compilers. The technique has recently been extended to dependently typed languages, but existing approaches have practical and theoretical shortcomings. We present a dependent-type-preserving translation into continuation-passing style (CPS). As improvements from previous work, our translation yields no administrative redexes, and its output can be typed using standard typing rules. This is achieved by defining an auxiliary translation that produces `let`-represented continuations in selected cases. The unique design makes the output of the translation partly look like A-normal form (ANF).

Keywords: CPS translation, ANF translation, dependent types

ACM Reference Format:
Youyou Cong, Hironori Kawazoe, and Hidehiko Masuhara. 2024. A Mostly CPS, Partly ANF Translation of Dependent Types. In *Proceedings of the 36th Symposium on Implementation and Application of Functional Languages (IFL '24).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Type-preserving compilation converts a well-typed source program to a well-typed target program through a series of type-preserving translations. The technique is a practical approach to building reliable compilers: it provides us a form of safety guarantees without requiring as much proof effort as full verification does [1].

In the past decade, there have been some notable advances in type-preserving compilation of dependently typed languages. The compiler passes considered so far include CPS translation [8, 11, 21], ANF translation [20], defunctionalization [18], closure conversion [7], and memory allocation [19]. It has been shown that all of these passes can be formulated as a type-preserving mapping if the source and/or target languages are designed appropriately.

This work is a continuation of Bowman et al. s [8] work on CPS translation of dependently typed languages. Specifically, we aim to address the following shortcomings of their translation.

1. The output of the translation contains *administrative redexes*. These redexes have to be eliminated through a second pass; otherwise the output would be slow to execute and hard to reason about.
2. The target language of the translation is equipped with a non-standard typing rule. This rule has to be proven not to break consistency of the language, which requires non-trivial reasoning.

To solve these issues, we define an auxiliary translation that, in some selected cases, yields a continuation expressed as a `let` expression. The auxiliary translation is essentially equivalent to what is called *colon translation* [16, 24], a traditional technique for avoiding administrative redexes. When tweaked to produce `let`-represented continuations, the translation further eliminates the need for a non-standard typing rule. An implication of this tweak is that the output of the translation is not fully CPS; it may contain parts that look like ANF [17].

In this paper, we report our outcomes so far, which include a formalization of the source and target languages (Sections 3 and 4), a definition of the CPS translation (Sections 5), and a sketch of our ongoing type preservation proof (Section 6). We also provide an informal description of our ideas (Section 2) as well as discussions of related and future work (Section 7 and 8).

2 Main Ideas

In this section, we give the reader a high-level idea of what is challenging with CPS translation of dependently typed languages and how we solve those challenges. Following previous work [8, 11], we use a non-bold blue sans-serif font to typeset the source language and a **bold red serif font** to typeset the target language. We also use the symbol $\div$ to mean a *computation translation*, which introduces a continuation, and $+$ to mean a *value translation*, which does not introduce a continuation.

2.1 Translating Terms into CPS

A CPS translation converts a term into a form that receives a continuation. As an example, let us look at the call-by-name CPS translation of snd e, which represents the second projection of a pair.

$$(\mathsf{snd}\ \mathsf{e})^{\div} = \boldsymbol{\lambda k}.\ \mathsf{e}^{\div}\ (\boldsymbol{\lambda y}.\ \mathbf{snd\ y\ k})$$

IFL '24, August 26–28, 2024, Nijmegen, The Netherlands
2024. https://doi.org/XXXXXXX.XXXXXXX

The result of the translation says: given a continuation $\mathbf{k}$, we evaluate e, and when we obtain the result $\mathbf{y}$, we take its second element and continue evaluation with $\mathbf{k}$.

The above translation contains an application of a CPS-translated term $\mathsf{e}^{\div}$ to a continuation. Such an application is called an *administrative redex*: it arises from the translation, not the source term. Administrative redexes are undesirable from a practical perspective, because they increase the number of reduction steps. They are also unpreferred from a theoretical point of view, as they complicate the reasoning of translated terms. Hence, in compilers such as Steele's [29] Rabbit, CPS translation is followed by a second pass that reduces administrative redexes.

Instead of taking the two-pass approach, one could also avoid generation of administrative redexes by optimizing the translation. This can be done by defining an auxiliary *colon translation* [16, 24], which translate a source term with respect to a target continuation. Below is the optimizing translation of second projection. To avoid confusion with the typing relation, we use a vertical bar to represent the colon translation.

$$
\begin{aligned}
&(\mathsf{snd}\ \mathsf{e})^{\div} \\
&= \boldsymbol{\lambda}\mathbf{k}.\ \mathsf{snd}\ \mathsf{e} \mid \mathbf{k} \\
&= \mathsf{e} \mid \boldsymbol{\lambda}\mathbf{y}.\ (\mathbf{snd}\ \mathbf{y})\ \mathbf{k} \\
&= \begin{cases} (\boldsymbol{\lambda}\mathbf{y}.\ (\mathbf{snd}\ \mathbf{y})\ \mathbf{k})\ \mathbf{e} & \text{if } \mathsf{e} \text{ is a value} \\ \mathbf{e}\ (\boldsymbol{\lambda}\mathbf{y}.\ (\mathbf{snd}\ \mathbf{y})\ \mathbf{k}) & \text{if } \mathsf{e} \text{ is not a value} \end{cases}
\end{aligned}
$$

As we can see, the translation produces different forms of output depending on whether e is a value or not. If e is a value, it *applies* the continuation $\boldsymbol{\lambda}\mathbf{y}.\ \mathbf{snd}\ \mathbf{y}\ \mathbf{k}$ to $\mathbf{e}$ (where $\mathbf{e}$ is a translation of e), which corresponds to returning the value $\mathbf{e}$. If e is not a value, it *passes* the continuation to $\mathbf{e}$, which corresponds to running the computation $\mathbf{e}$. In general, a colon translation places the continuation directly to the "right place", rather than letting it go to the right place via administrative reductions.

2.2 Typing Result of Translation

For a CPS translation to be type preserving, it must satisfy the following property: if we have $\mathsf{e} : \mathsf{A}$ in the source, then we have $\mathsf{e}^{\div} : (\mathsf{A}^{+} \to \boldsymbol{\omega}) \to \boldsymbol{\omega}$ in the target, where $\boldsymbol{\omega}$ is an answer type. This property holds trivially for second projection $\mathsf{snd}\ \mathsf{e}$ when e has a non-dependent pair type $\mathsf{A} \times \mathsf{B}$, but not when e has a dependent pair type $\Sigma \mathsf{x} : \mathsf{A}.\ \mathsf{B}$ and is a non-value. To see the challenge, let us recall that the second projection of a dependent pair has a type that depends on the first element of the pair. This means, the source term $\mathsf{snd}\ \mathsf{e}$ to be CPS translated has type $\mathsf{B}\ [\mathsf{fst}\ \mathsf{e}/\mathsf{x}]$, and the target term $\mathbf{snd}\ \mathbf{y}$ in the reuslt of the translation has type $(\mathsf{B}^{+}\ [\mathbf{fst}\ \mathbf{y}/\mathbf{x}] \to \bot) \to \bot$. Now, notice that the latter type refers to the variable $\mathbf{y}$, which is introduced by the translation. This variable cannot appear in the type of the continuation $\mathbf{k}$, because it is calculated from the type of the source term $\mathsf{snd}\ \mathsf{e}$. Indeed, when instantiating the type preservation statement to second projection, we can easily see that $\mathbf{k}$ must have type $\mathsf{B}^{+}\ [(\mathsf{snd}\ \mathsf{e})^{\div}/\mathbf{x}] \to \boldsymbol{\omega}$. What this implies is that the application $\mathbf{snd}\ \mathbf{y}\ \mathbf{k}$ is not well-typed.

This kind of type mismatch occurs not only in the translation of second projection, but also in, *e.g.*, the call-by-value translation of application. More generally, it arises in any cases where the translation yields a term whose type depends on a variable representing the argument of a continuation ($\mathbf{y}$ in the case of second projection).

Fortunately, if the source language is pure, it is possible to prove that the application $\mathbf{snd}\ \mathbf{y}\ \mathbf{k}$ is well-typed. The well-typedness relies on the fact that, if e is a pure term, the continuation passed to $\mathsf{e}^{\div}$ must receive a unique value. This unique value correponds to the result of evaluating e, and it can be obtained by running a CPS-translated term $\mathsf{e}^{\div}$ with the identity continuation $\mathbf{id}$ (in the case of a non-optimizing translation), or by colon-translating a source term with the identity continuation (in the case of an optimizing translation).

The above fact leaves us with two questions.

1. How to ensure that the unique value received by a continuation can always be obtained in a type-safe way
2. How to make the knowledge about the unique value available for use in the type preservation proof

These questions were answered by Bowman et al. [8] in the following way.

1. Replace the fixed answer type of the translation with a polymorphic one. This makes the use of the identity continuation type-safe.

$$\mathsf{e} : \mathsf{A} \quad \Rightarrow \quad \mathsf{e}^{\div} : \boldsymbol{\Pi}\boldsymbol{\gamma} : *_{\mathsf{A}}.\ (\mathsf{A}^{+} \to \boldsymbol{\gamma}) \to \boldsymbol{\gamma}$$

2. Define a special typing rule [T-Cont] in the target language. This introduces into the typing environment a definition binding the unique value to the argument of a continuation.

$$\frac{\boldsymbol{\Gamma} \vdash \mathbf{e}_1 : \boldsymbol{\Pi}\boldsymbol{\alpha} : *_{\mathbf{A}}.\ (\mathbf{A} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha} \qquad \boldsymbol{\Gamma}, \mathbf{y} = \mathbf{e}_1\ \mathbf{A}\ \mathbf{id} : \mathbf{A} \vdash \mathbf{e}_2 : \mathbf{B}}{\boldsymbol{\Gamma} \vdash \mathbf{e}_1\ \mathbf{B}\ (\boldsymbol{\lambda}\mathbf{y}.\ \mathbf{e}_2) : \mathbf{B}}\ [\text{T-Cont}]$$

These answers solve the type mismatch in the application $\mathbf{snd}\ \mathbf{y}\ \mathbf{k}$, but there is room for improvement. In a dependently typed language, where consistency (*i.e.*, non-inhabitance of the bottom type $\bot$) is often critical, adding a new typing rule is a delicate task that requires careful justification. The [T-Cont] rule of Bowman et al. has been proven safe, but the proof is no simpler than the type preservation proof, involving sophisticated reasoning based on parametricity.

To reduce the justification effort, we give a different answer to the second question: whenever we need the information about the argument of a continuation, we represent that

Universes	U, V	::=	$* \mid \Box$
Kinds	K	::=	$* \mid \Pi\alpha : K.\, K \mid \Pi x : A.\, K$
Types	A, B	::=	$\alpha \mid \lambda\alpha : K.\, A \mid \lambda x : A.\, A$
			$\mid\ A\ A \mid A\ e$
			$\mid\ \Pi\alpha : K.\, A \mid \Pi x : A.\, A$
			$\mid\ \Sigma x : A.\, A$
			$\mid\ \texttt{let}\ \alpha = A : K\ \texttt{in}\ A$
			$\mid\ \texttt{let}\ x = e : A\ \texttt{in}\ A$
Terms	e	::=	$v \mid p$
Values	v	::=	$\lambda\alpha : K.\, e \mid \lambda x : A.\, e$
			$\mid\ \langle e, e\rangle\ \texttt{as}\ A$
Non-values	p	::=	$x \mid e\ A \mid e\ e$
			$\mid\ \texttt{fst}\ e \mid \texttt{snd}\ e$
			$\mid\ \texttt{let}\ \alpha = A : K\ \texttt{in}\ e$
			$\mid\ \texttt{let}\ x = e : A\ \texttt{in}\ e$
Environments	Γ	::=	$\cdot \mid \Gamma, \alpha : K \mid \Gamma, x : A$
			$\mid\ \Gamma, \alpha = A : K \mid \Gamma, x = e : A$

Figure 1. CC Syntax

$$\Gamma \vdash z \ \triangleright_\delta\ M \text{ where } z = M : T \in \Gamma$$
$$\Gamma \vdash (\lambda z : T.\, M)\ N \ \triangleright_\beta\ M\,[N/z]$$
$$\Gamma \vdash \texttt{fst}\ (\langle e_1, e_2\rangle\ \texttt{as}\ A) \ \triangleright_{\pi_1}\ e_1$$
$$\Gamma \vdash \texttt{snd}\ (\langle e_1, e_2\rangle\ \texttt{as}\ A) \ \triangleright_{\pi_2}\ e_2$$
$$\Gamma \vdash \texttt{let}\ z = M : T\ \texttt{in}\ N \ \triangleright_\zeta\ M\,[N/z]$$

$$\frac{}{\Gamma \vdash M \equiv M}\ (\equiv\text{-Refl}) \qquad \frac{\Gamma \vdash M_1 \equiv M_2}{\Gamma \vdash M_2 \equiv M_1}\ (\equiv\text{-Sym})$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 \quad \Gamma \vdash M_2 \equiv M_3}{\Gamma \vdash M_1 \equiv M_3}\ (\equiv\text{-Trans})$$

$$\frac{\Gamma \vdash M_1 \triangleright^* N \quad \Gamma \vdash M_2 \triangleright^* N}{\Gamma \vdash M_1 \equiv M_2}\ (\equiv\text{-}\triangleright^*)$$

Figure 2. CC Reduction and Equivalence

continuation using a `let` expression. For instance, we translate the second projection of a non-value e in the following way. Note that the colon translation takes three arguments, among which the second one is used to instantiate the polymorphic answer type.

$$\begin{aligned}&(\texttt{snd}\ e)^\div\\ &= \boldsymbol{\lambda k}.\, \texttt{snd}\ e \mid \boldsymbol{\alpha} \mid \mathbf{k}\\ &= \mathbf{let\ y} = e \mid \boldsymbol{\Sigma x} : A^\div.\, B^\div \mid \mathbf{id\ in\ snd\ y}\ \boldsymbol{\alpha}\ \mathbf{k}\end{aligned}$$

Observe that the continuation $\boldsymbol{\lambda y}.\, \mathbf{snd\ y\ k}$ is now represented as $\mathbf{let\ y} = [.]\ \mathbf{in\ snd\ y}\ \boldsymbol{\alpha}\ \mathbf{k}$, where $[.]$ denotes a hole. The expression inside the hole is the target language counterpart of the value of e, and it is bound explicitly to the variable $\mathbf{y}$. When typing the `let` expression, we use the standard typing rule [Let] given by Severi and Poll [28], which allows us to type the body $\mathbf{snd\ y}\ \boldsymbol{\alpha}\ \mathbf{k}$ using the definition $\mathbf{y} = e \mid \boldsymbol{\Sigma x} : A^\div.\, B^\div \mid \mathbf{id}$.

$$\frac{\boldsymbol{\Gamma}, \mathbf{x} = \mathbf{M} : \mathbf{A} \vdash \mathbf{N} : \mathbf{B}}{\boldsymbol{\Gamma} \vdash \mathbf{let\ x} = \mathbf{M} : \mathbf{A\ in\ N} : \mathbf{B}\,[\mathbf{M}/\mathbf{y}]}\ [\text{Let}]$$

In general, if we use a `let` to represent any continuation whose typing requires [T-Cont], we are able to preserve types without non-standard rules.

3 Source Language

The source language CC of our CPS translation is an extension of the Calculus of Constructions (CoC) [13] with dependent pairs and `let` expressions. In this section, we give the syntax, reduction, equivalence, and typing of CC.

3.1 Syntax

Figure 1 shows the syntax of CC. The language has two universes: $*$, which is the type of types, and $\Box$, which is the type of kinds. Function types are defined in both kind and type categories, whereas pair types are only defined in the type category. Terms are stratified into values and non-values. While the stratification is not necessary for specifying the semantics of CC, it is useful for defining an optimizing CPS translation. Lastly, environments consist of variable bindings ($\alpha : K$ and $x : A$) introduced by functions, as well as definitions [28] ($\alpha = A : K$ and $x = e : A$) introduced by `let` expressions.

As the figure shows, there are syntactic constructs that have variations differing in which categories their components belong to. To avoid having to define reduction and typing rules for each variation, we use the following notational convention throughout the paper.

$$\begin{aligned}M, N &= \text{a kind } K, \text{ type } A, \text{ or term } e\\ S, T &= \text{a universe } U, \text{ kind } K, \text{ or type } A\\ z &= \text{a type variable } \alpha \text{ or term variable } x\end{aligned}$$

We also use "expression" as an umbrella word for kinds, types, and terms.

3.2 Reduction and Equivalence

Figure 2 top shows the reduction rules of CC. The judgment carries an environment Γ, which is used by the δ rule to replace a variable with an expression according to a definition. Other rules define how functions, pairs, and `let` expressions are eliminated, and they are completely standard.

We define $\Gamma \vdash M \triangleright^* N$ as the reflexive, transitive, and compatible closure of $\Gamma \vdash M \triangleright N$.

Figure 2 bottom shows the equivalence rules of CC. The first three rules define equivalence as a reflexive, symmetric, and transitive relation. The last rule defines equivalence in

$$\frac{}{\vdash \cdot}\ (\textsc{Empty}) \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T : U \quad z \notin dom(\Gamma)}{\vdash \Gamma, z : T}\ (\textsc{Extend})$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash M : T \quad z \notin dom(\Gamma)}{\vdash \Gamma, z = M : T}\ (\textsc{ExtendDef})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash *}\ (\textsc{Star}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash \Box}\ (\textsc{Box})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \Box}\ (\textsc{Ax}) \qquad \frac{z : T \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash z : T}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash S : U \quad \Gamma, z : S \vdash T : V}{\Gamma \vdash (\Pi z : S.\, T) : V}\ (\textsc{Pi})$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash (\Sigma x : A.\, B) : *}\ (\textsc{Sigma})$$

$$\frac{\Gamma, z : S \vdash M : T}{\Gamma \vdash (\lambda z : S.\, M) : (\Pi z : S.\, T)}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash M : (\Pi z : S.\, T) \quad \Gamma \vdash N : S}{\Gamma \vdash M\ N : T\,[N/z]}\ (\textsc{App})$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B\,[e_1/x]}{\Gamma \vdash (\langle e_1, e_2 \rangle \text{ as } \Sigma x : A.\, B) : (\Sigma x : A.\, B)}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash e : (\Sigma x : A.\, B)}{\Gamma \vdash \text{fst } e : A}\ (\textsc{Fst}) \qquad \frac{\Gamma \vdash e : (\Sigma x : A.\, B)}{\Gamma \vdash \text{snd } e : B\,[\text{fst } e/x]}\ (\textsc{Snd})$$

$$\frac{\Gamma \vdash M : S \quad \Gamma, z = M : S \vdash N : T}{\Gamma \vdash \text{let } z = M : S \text{ in } N : T\,[M/z]}\ (\textsc{Let})$$

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash T : U \quad \Gamma \vdash S \equiv T}{\Gamma \vdash M : T}\ (\textsc{Conv})$$

Figure 3. CC Typing

terms of reduction: if two expressions reduce to the same expression, they are considered equivalent.

3.3 Typing

Figure 3 shows the typing rules of CC. The rule (Pi) allows the variable z to appear in the co-domain B, and similarly for (Sigma). The rule (Let) makes the definition $z = M : S$ available for use in the typing of the body N. The last rule

Universes	$\mathbf{U}, \mathbf{V}$	::=	$*_A \mid \Box$
Kinds	$\mathbf{K}$	::=	$*_R \mid *_A \mid *_C$
		$\mid$	$\Pi \alpha : \mathbf{K}.\, \mathbf{K} \mid \Pi x : \mathbf{R}.\, \mathbf{K}$
Root Types	$\mathbf{R}$	::=	$\Pi \gamma : *_A.\, (\mathbf{A} \to \gamma) \to \gamma$
Value Types	$\mathbf{A}, \mathbf{B}$	::=	$\alpha \mid \lambda \alpha : \mathbf{K}.\, \mathbf{A} \mid \lambda x : \mathbf{R}.\, \mathbf{A}$
		$\mid$	$\mathbf{A}\ \mathbf{A} \mid \mathbf{A}\ \mathbf{r}$
		$\mid$	$\Pi \alpha : \mathbf{K}.\, \mathbf{R} \mid \Pi x : \mathbf{R}.\, \mathbf{R}$
		$\mid$	$\Sigma x : \mathbf{R}.\, \mathbf{R}$
		$\mid$	$\mathbf{let}\ \alpha = \mathbf{A} : \mathbf{K}\ \mathbf{in}\ \mathbf{A}$
		$\mid$	$\mathbf{let}\ x = \mathbf{r} : \mathbf{R}\ \mathbf{in}\ \mathbf{A}$
Answer Types	ω_θ	::=	$(\theta = \mathbf{c})\ \mathbf{A} \mid (\theta = \mathbf{o})\ \gamma$
Roots	$\mathbf{r}$	::=	$\lambda \gamma : *_A.\, \lambda \mathbf{k} : \mathbf{A} \to \gamma.\, \mathbf{a}$
Values	$\mathbf{v}$	::=	$\mathbf{y} \mid \lambda \alpha : \mathbf{K}.\, \mathbf{r} \mid \lambda x : \mathbf{R}.\, \mathbf{r}$
		$\mid$	$\langle \mathbf{r}, \mathbf{r} \rangle\ \mathbf{as}\ \mathbf{A} \mid \mathbf{a}_c$
Non-values	$\mathbf{p}$	::=	$x \mid \mathbf{v}\ \mathbf{A} \mid \mathbf{v}\ \mathbf{r}$
		$\mid$	$\mathbf{fst}\ \mathbf{v} \mid \mathbf{snd}\ \mathbf{v}$
Answers	$\mathbf{a}_\theta$	::=	$\kappa_\theta\ \mathbf{v} \mid \mathbf{p}\ \omega_\theta\ \kappa_\theta$
		$\mid$	$\mathbf{let}\ \alpha = \mathbf{A} : \mathbf{K}\ \mathbf{in}\ \mathbf{a}_\theta$
		$\mid$	$\mathbf{let}\ x = \mathbf{r} : \mathbf{R}\ \mathbf{in}\ \mathbf{a}_\theta$
		$\mid$	$\mathbf{let}\ \mathbf{y} = \mathbf{v} : \mathbf{A}\ \mathbf{in}\ \mathbf{a}_\theta$
Continuations	κ_θ	::=	$(\theta = \mathbf{c})\ \mathbf{id}_\mathbf{A}$
		$\mid$	$(\theta = \mathbf{o})\ \mathbf{k}$
		$\mid$	$\lambda \mathbf{y} : \mathbf{A}.\, \mathbf{a}_\theta$
Annotations	θ	::=	$\mathbf{c} \mid \mathbf{o}$
Environments	Γ	::=	$\cdot \mid \Gamma, \alpha : \mathbf{K} \mid \Gamma, x : \mathbf{R}$
		$\mid$	$\Gamma, \alpha = \mathbf{A} : \mathbf{K} \mid \Gamma, x = \mathbf{r} : \mathbf{R}$
		$\mid$	$\Gamma, \mathbf{y} = \mathbf{v} : \mathbf{A}$

Figure 4. CC^k Syntax

(Conv) converts the type of an expression to a syntactically different but semantically equivalent type.

4 Target Language

The target language CC^k of our CPS translation is roughly a subset of CC conforming to certain syntactic constraints[1]. In this section, we go through the specification while highlighting what are the constraints and how they are imposed.

4.1 Syntax

Figure 4 shows the syntax of CC^k. The language has five groups of expressions: roots, answers, values, non-values, and continuations. Roots are the output of the computation translation, and answers are the output of the answer translation (or, equivalently, the result of running a CPS-translated term with an answer type and a continuation). In the categories of answer types and continuations, γ and $\mathbf{k}$ are fixed variables introduced by the translation.

[1]We could use the plain CoC (with dependent `let`) as the target language, but we choose to define a precise target language because it allows us to discuss the inverse of CPS translation, which is useful for proving compiler correctness [27].

$$
\begin{array}{rcl}
\Gamma \vdash \mathbf{z} & \triangleright_\delta & \mathbf{M} \text{ where } \mathbf{z} = \mathbf{M} : \mathbf{T} \in \Gamma \\
\Gamma \vdash \mathbf{y} & \triangleright_\delta & \mathbf{v} \text{ where } \mathbf{y} = \mathbf{v} : \mathbf{A} \in \Gamma \\
\Gamma \vdash (\lambda \mathbf{z} : \mathbf{T}.\, \mathbf{A})\ \mathbf{M} & \triangleright_\beta & \mathbf{A}\,[\mathbf{M}/\mathbf{z}] \\
\Gamma \vdash (\lambda \mathbf{z} : \mathbf{T}.\, \lambda \boldsymbol{\gamma} : *_{\mathrm{A}}.\, \lambda \mathbf{k} : \mathbf{A} \to \boldsymbol{\gamma}.\, \mathbf{a})\ \mathbf{M}\ \boldsymbol{\omega}\ \boldsymbol{\kappa} & \triangleright_\beta & \mathbf{a}\,[\mathbf{M}/\mathbf{z}, \boldsymbol{\omega}/\boldsymbol{\gamma}, \boldsymbol{\kappa}/\mathbf{k}] \\
\Gamma \vdash (\lambda \mathbf{y} : \mathbf{A}.\, \mathbf{a})\ \mathbf{v} & \triangleright_\beta & \mathbf{a}\,[\mathbf{v}/\mathbf{y}] \\
\Gamma \vdash \mathbf{fst}\ (\langle \lambda \boldsymbol{\gamma} : *_{\mathrm{A}}.\, \lambda \mathbf{k} : \mathbf{A} \to \boldsymbol{\gamma}.\, \mathbf{a}, \mathbf{r} \rangle\ \mathbf{as}\ \mathbf{B})\ \boldsymbol{\omega}\ \boldsymbol{\kappa} & \triangleright_{\pi_1} & \mathbf{a}\,[\boldsymbol{\omega}/\boldsymbol{\gamma}, \boldsymbol{\kappa}/\mathbf{k}] \\
\Gamma \vdash \mathbf{snd}\ (\langle \mathbf{r}, \lambda \boldsymbol{\gamma} : *_{\mathrm{A}}.\, \lambda \mathbf{k} : \mathbf{A} \to \boldsymbol{\gamma}.\, \mathbf{a} \rangle\ \mathbf{as}\ \mathbf{B})\ \boldsymbol{\omega}\ \boldsymbol{\kappa} & \triangleright_{\pi_2} & \mathbf{a}\,[\boldsymbol{\omega}/\boldsymbol{\gamma}, \boldsymbol{\kappa}/\mathbf{k}] \\
\Gamma \vdash \mathbf{let}\ \mathbf{z} = \mathbf{M} : \mathbf{T}\ \mathbf{in}\ \mathbf{A} & \triangleright_\zeta & \mathbf{A}\,[\mathbf{M}/\mathbf{z}] \\
\Gamma \vdash \mathbf{let}\ \mathbf{z} = \mathbf{M} : \mathbf{T}\ \mathbf{in}\ \mathbf{a} & \triangleright_\zeta & \mathbf{a}\,[\mathbf{M}/\mathbf{z}] \\
\Gamma \vdash \mathbf{let}\ \mathbf{y} = \mathbf{v} : \mathbf{A}\ \mathbf{in}\ \mathbf{a} & \triangleright_\zeta & \mathbf{a}\,[\mathbf{v}/\mathbf{y}]
\end{array}
$$

$$
\frac{}{\Gamma \vdash \lambda \boldsymbol{\gamma} : *_{\mathrm{A}}.\, \lambda \mathbf{k} : \mathbf{A} \to \boldsymbol{\gamma}.\, \mathbf{p}\ \boldsymbol{\gamma}\ \mathbf{k} \equiv \mathbf{p}}\ [\equiv\text{-}\eta] \qquad \frac{\Gamma \vdash \mathbf{M}_1 \equiv \mathbf{M}_2 \qquad \Gamma \vdash \mathbf{N}_1 \equiv \mathbf{N}_2}{\Gamma \vdash \mathbf{M}_1\ \mathbf{N}_1 \equiv \mathbf{M}_2\ \mathbf{N}_2}\ [\equiv\text{-App}]
$$

Figure 5. CC^k Reduction and Equivalence (excerpt)

$$
\frac{\vdash \Gamma \quad \Gamma \vdash \mathbf{M} : \mathbf{T} \quad \mathbf{z} \notin dom(\Gamma)}{\vdash \Gamma, \mathbf{z} = \mathbf{M} : \mathbf{T}}\ [\textsc{ExtendDef}] \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \mathbf{v} : \mathbf{A} \quad \mathbf{y} \notin dom(\Gamma)}{\vdash \Gamma, \mathbf{y} = \mathbf{v} : \mathbf{A}}\ [\textsc{ExtendDef}_c]
$$

$$
\frac{\Gamma \vdash \mathbf{S} : \mathbf{U} \quad \Gamma, \mathbf{z} : \mathbf{S} \vdash \mathbf{T} : \mathbf{V}}{\Gamma \vdash (\Pi \mathbf{z} : \mathbf{S}.\, \mathbf{T}) : \mathbf{V}}\ [\textsc{Pi}] \qquad \frac{\Gamma \vdash \mathbf{R}_1 : *_{\mathrm{R}} \quad \Gamma, \mathbf{x} : \mathbf{R}_1 \vdash \mathbf{R}_2 : *_{\mathrm{R}}}{\Gamma \vdash (\Sigma \mathbf{x} : \mathbf{R}_1.\, \mathbf{R}_2) : *_{\mathrm{R}}}\ [\textsc{Sigma}]
$$

$$
\frac{\Gamma, \mathbf{z} : \mathbf{T} \vdash \mathbf{A} : \mathbf{K}}{\Gamma \vdash (\lambda \mathbf{z} : \mathbf{T}.\, \mathbf{A}) : (\Pi \mathbf{z} : \mathbf{T}.\, \mathbf{K})}\ [\textsc{Abs}_\square] \qquad \frac{\Gamma \vdash \mathbf{A} : (\Pi \mathbf{z} : \mathbf{T}.\, \mathbf{K}) \quad \Gamma \vdash \mathbf{M} : \mathbf{T}}{\Gamma \vdash \mathbf{A}\ \mathbf{M} : \mathbf{K}\,[\mathbf{M}/\mathbf{z}]}\ [\textsc{App}_\square] \qquad \frac{\Gamma, \mathbf{z} = \mathbf{M} : \mathbf{T} \vdash \mathbf{A} : \mathbf{K}}{\Gamma \vdash \mathbf{let}\ \mathbf{z} = \mathbf{M} : \mathbf{T}\ \mathbf{in}\ \mathbf{A} : \mathbf{K}\,[\mathbf{M}/\mathbf{z}]}\ [\textsc{Let}_\square]
$$

$$
\frac{\Gamma \vdash \mathbf{A} : *_{\mathrm{A}}}{\Gamma \vdash \Pi \boldsymbol{\gamma} : *_{\mathrm{A}}.\, (\mathbf{A} \to \boldsymbol{\gamma}) \to \boldsymbol{\gamma} : *_{\mathrm{R}}}\ [\textsc{RootTy}] \qquad \frac{\mathbf{z} : \mathbf{T} \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash \mathbf{z} : \mathbf{T}}\ [\textsc{Var}] \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \mathbf{A} : *_{\mathrm{A}}}{\Gamma \mid \mathbf{A} \vdash \boldsymbol{\gamma} : *_{\mathrm{A}}}\ [\textsc{AVar}]
$$

$$
\frac{\Gamma \vdash \mathbf{A} : *_{\mathrm{A}} \quad \Gamma \mid \Theta \vdash \boldsymbol{\omega} : *_{\mathrm{A}}}{\Gamma \mid \Theta \vdash \mathbf{A} \to \boldsymbol{\omega} : *_{\mathrm{C}}}\ [\textsc{ContArrow}] \qquad \frac{\Gamma \vdash \mathbf{M} : \mathbf{S} \quad \Gamma \vdash \mathbf{S} \equiv \mathbf{T}}{\Gamma \vdash \mathbf{M} : \mathbf{T}}\ [\textsc{Conv}]
$$

$$
\frac{\Gamma \mid \mathbf{A} \vdash \mathbf{a} : \boldsymbol{\alpha}}{\Gamma \vdash \lambda \boldsymbol{\gamma} : *_{\mathrm{A}}.\, \lambda \mathbf{k} : \mathbf{A} \to \boldsymbol{\gamma}.\, \mathbf{a}_o : \Pi \boldsymbol{\gamma} : *_{\mathrm{A}}.\, (\mathbf{A} \to \boldsymbol{\gamma}) \to \boldsymbol{\gamma}}\ [\textsc{Root}]
$$

$$
\frac{\Gamma \mid \_ \vdash \mathbf{a}_c : \mathbf{A}}{\Gamma \vdash \mathbf{a}_c : \mathbf{A}}\ [\textsc{ClosedAns}_*] \qquad \frac{\Gamma, \mathbf{z} : \mathbf{T} \vdash \mathbf{r} : \mathbf{R}}{\Gamma \vdash (\lambda \mathbf{z} : \mathbf{T}.\, \mathbf{r}) : (\Pi \mathbf{z} : \mathbf{T}.\, \mathbf{R})}\ [\textsc{Abs}_*] \qquad \frac{\Gamma \vdash \mathbf{r}_1 : \mathbf{R}_1 \quad \Gamma \vdash \mathbf{r}_2 : \mathbf{R}_2\,[\mathbf{r}_1/\mathbf{x}]}{\Gamma \vdash (\langle \mathbf{r}_1, \mathbf{r}_2 \rangle\ \mathbf{as}\ \Sigma \mathbf{x} : \mathbf{R}_1.\, \mathbf{R}_2) : (\Sigma \mathbf{x} : \mathbf{R}_1.\, \mathbf{R}_2)}\ [\textsc{Pair}]
$$

$$
\frac{\Gamma \vdash \mathbf{v} : (\Pi \mathbf{z} : \mathbf{T}.\, \mathbf{R}) \quad \Gamma \vdash \mathbf{M} : \mathbf{T}}{\Gamma \vdash \mathbf{v}\ \mathbf{M} : \mathbf{R}\,[\mathbf{M}/\mathbf{z}]}\ [\textsc{App}_*] \qquad \frac{\Gamma \vdash \mathbf{v} : (\Sigma \mathbf{x} : \mathbf{R}_1.\, \mathbf{R}_2)}{\Gamma \vdash \mathbf{fst}\ \mathbf{v} : \mathbf{R}_1}\ [\textsc{Fst}] \qquad \frac{\Gamma \vdash \mathbf{v} : (\Sigma \mathbf{x} : \mathbf{R}_1.\, \mathbf{R}_2)}{\Gamma \vdash \mathbf{snd}\ \mathbf{v} : \mathbf{R}_2\,[\mathbf{fst}\ \mathbf{v}/\mathbf{x}]}\ [\textsc{Snd}]
$$

$$
\frac{\Gamma \mid \Theta \vdash \kappa_\theta : \mathbf{A} \to \omega_\theta \quad \Gamma \vdash \mathbf{v} : \mathbf{A}}{\Gamma \mid \Theta \vdash \kappa_\theta\ \mathbf{v} : \omega_\theta}\ [\textsc{Return}] \qquad \frac{\Gamma \vdash \mathbf{p} : \Pi \boldsymbol{\gamma} : *_{\mathrm{A}}.\, (\mathbf{A} \to \boldsymbol{\gamma}) \to \boldsymbol{\gamma} \quad \Gamma \mid \Theta \vdash \omega_\theta : *_{\mathrm{A}} \quad \Gamma \mid \Theta \vdash \kappa_\theta : \mathbf{A} \to \omega_\theta}{\Gamma \mid \Theta \vdash \mathbf{p}\ \omega_\theta\ \kappa_\theta : \omega_\theta}\ [\textsc{Run}]
$$

$$
\frac{\Gamma, \mathbf{z} = \mathbf{M} : \mathbf{T} \mid \Theta \vdash \mathbf{a}_\theta : \omega_\theta}{\Gamma \mid \Theta \vdash \mathbf{let}\ \mathbf{z} = \mathbf{M} : \mathbf{T}\ \mathbf{in}\ \mathbf{a}_\theta : \boldsymbol{\omega}\,[\mathbf{M}/\mathbf{z}]}\ [\textsc{Let}_\theta] \qquad \frac{\Gamma, \mathbf{y} = \mathbf{v} : \mathbf{A} \mid \Theta \vdash \mathbf{a}_\theta : \omega_\theta}{\Gamma \mid \Theta \vdash \mathbf{let}\ \mathbf{y} = \mathbf{v} : \mathbf{A}\ \mathbf{in}\ \mathbf{a}_\theta : \omega_\theta\,[\mathbf{v}/\mathbf{y}]}\ [\textsc{Let}_c]
$$

$$
\frac{\vdash \Gamma \quad \Gamma \vdash \mathbf{A} : *_{\mathrm{A}}}{\Gamma \mid \_ \vdash \mathbf{id}_{\mathbf{A}} : \mathbf{A} \to \mathbf{A}}\ [\textsc{IdCont}] \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \mathbf{A} : *_{\mathrm{A}}}{\Gamma \mid \mathbf{A} \vdash \mathbf{k} : \mathbf{A} \to \boldsymbol{\gamma}}\ [\textsc{ContVar}] \qquad \frac{\Gamma, \mathbf{y} : \mathbf{A} \mid \Theta \vdash \mathbf{a}_\theta : \omega_\theta}{\Gamma \mid \Theta \vdash \lambda \mathbf{y} : \mathbf{A}.\, \mathbf{a}_\theta : \mathbf{A} \to \omega_\theta}\ [\textsc{ExtendCont}]
$$

Figure 6. CC^k Typing (exerpt)

An important observation we can make here is that answers and continuations carry an annotation θ[2]. These annotations are borrowed from existing work on optimizing CPS translations of control effects [5, 6], and are used to precisely characterize the image of the CPS translation. When $\theta = \mathbf{c}$, we say the answer or continuation is *closed*, in the sense that it does not have free occurrences of the answer type variable γ or continuation variable $\mathbf{k}$. When $\theta = \mathbf{o}$, we say the answer or continuation is *open*, in that it has free occurrences of γ and $\mathbf{k}$. Closed answers are identified with values, and they appear in the definition of `let` expressions $\mathbf{let}\ \mathbf{y} = \mathbf{v} : \mathbf{A}\ \mathbf{in}\ \mathbf{a}_\theta$ introduced by the translation.

As we did for the source language, we define the following notational convention to keep the specification concise.

$$\begin{aligned} \mathbf{M}, \mathbf{N} &= \text{a value kind } \mathbf{K}\text{, value type } \mathbf{A}\text{, or root term } \mathbf{r} \\ \mathbf{S}, \mathbf{T} &= \text{a universe } \mathbf{U}\text{, value kind } \mathbf{K}\text{, or root type } \mathbf{R} \\ \mathbf{z} &= \text{a type variable } \alpha \text{ or term variable } \mathbf{x} \end{aligned}$$

4.2 Reduction and Equivalence

Figure 5 top shows the reduction rules of CC^k. The δ rule has two variations, among which the first one reduces a redex arising from the source program and the second one reduces a redex arising from the translation. The second β rule and the two π rules perform substitutions for the answer type variable γ and continuation variable $\mathbf{k}$. These substitutions correspond to administrative reductions, and performing them as part of β and π rules help us establish a one-to-one correspondence between source and target reductions [5, 6]. The third β rule takes care of an application of a continuation to a value.

Figure 5 bottom shows selected equivalence rules of CC^k. Other than the four rules we have in CC, there is a rule defining η equivalence on roots. We also have congruence rules that allow us to regard expressions consisting of equivalent subexpressions as equivalent, although we do not include all of them in the figure due to space reasons. These additional rules are required for proving type preservation.

4.3 Typing

Figure 6 shows selected typing rules of CC^k[3]. Most rules use the standard typing judgment, but the rules for answers and continuations use a judgment that has an extra component Θ. This component represents the typing assumptions introduced by root abstractions $\lambda\gamma$ and $\lambda\mathbf{k}$. More precisely, when the answer or continuation is open, Θ is a value type $\mathbf{A}$ representing the domain of the continuation $\mathbf{k}$; this is sufficient because the return type of $\mathbf{k}$ must be γ and γ has type $*_{\mathbf{A}}$.

[2]For readability, we elide annotations when they are not essential.
[3]Omitted rules include environment extension, kind well-formedness, and conversion.

$$\frac{}{\vdash \cdot \rightsquigarrow^+ \cdot}$$

$$\frac{\vdash \Gamma \rightsquigarrow^+ \mathbf{\Gamma} \quad \Gamma \vdash \mathrm{K} : \Box \rightsquigarrow^+ \mathbf{K}}{\vdash \Gamma, \alpha : \mathrm{K} \rightsquigarrow^+ \mathbf{\Gamma}, \boldsymbol{\alpha} : \mathbf{K}}$$

$$\frac{\vdash \Gamma \rightsquigarrow^+ \mathbf{\Gamma} \quad \Gamma \vdash \mathrm{A} : * \rightsquigarrow^\div \mathbf{R}}{\vdash \Gamma, \mathrm{x} : \mathrm{A} \rightsquigarrow^+ \mathbf{\Gamma}, \mathbf{x} : \mathbf{R}}$$

$$\frac{\vdash \Gamma \rightsquigarrow^+ \mathbf{\Gamma} \quad \Gamma \vdash \mathrm{A} : \mathrm{K} \rightsquigarrow^+ \mathbf{A} \quad \Gamma \vdash \mathrm{K} : \Box \rightsquigarrow^+ \mathbf{K}}{\vdash \Gamma, \alpha = \mathrm{A} : \mathrm{K} \rightsquigarrow^+ \mathbf{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \mathbf{K}}$$

$$\frac{\vdash \Gamma \rightsquigarrow^+ \mathbf{\Gamma} \quad \Gamma \vdash \mathrm{e} : \mathrm{A} \rightsquigarrow^\div \mathbf{r} \quad \Gamma \vdash \mathrm{A} : * \rightsquigarrow^\div \mathbf{R}}{\vdash \Gamma, \mathrm{x} = \mathrm{e} : \mathrm{A} \rightsquigarrow^+ \mathbf{\Gamma}, \mathbf{x} = \mathbf{r} : \mathbf{R}}$$

Figure 7. CPS Translation of Typing Environments

When the answer or continuation is closed, Θ carries no information (_). A similar judgement has previously appeared in Barthe et al.'s [4] CPS translation of the CoC.

5 CPS Translation

We now define a call-by-name CPS translation of CC. Our design principle is to combine existing techniques, but with a twist.

- Following Danvy and Nielsen [16], we use an auxiliary translation of terms to avoid administrative redexes.
- Following Bowman et al. [8], we make the answer type polymorphic so that we can obtain the value of any CPS-translated term.
- As our own idea, we use a `let` expression to represent continuations at selected places and thus restore type preservation.

Let us begin with the translation of environments, universes, kinds, and types (Figures 7 and 8). The translation is defined on the typing derivation, not on the syntax. This is because we need the information of types to annotate continuation variables and their arguments in the translation of terms. For types, we have two kinds of translations: a *computation translation* $\div$ and a *value translation* $+$. The former essentially corresponds to double negation, with the consequence being a polymorphic answer type. The latter keeps the top-level shape of the original type. Note that, since we are modelling a call-by-name semantics, we apply the computation translation to the domain of function types, the two components of pair types, and the right-hand side of definitions.

We next look at the translation of terms. Again, we have a computation translation that introduces answer type and continuation abstractions, and a value translation that does not introduce those abstractions. In addition to these, we

$$\frac{}{\Gamma \vdash * \rightsquigarrow^{+} \mathbf{*}} \qquad \frac{}{\Gamma \vdash \Box \rightsquigarrow^{+} \mathbf{\Box}}$$

$$\frac{}{\Gamma \vdash * : \Box \rightsquigarrow^{+} \mathbf{*}}$$

$$\frac{\Gamma \vdash K_1 : \Box \rightsquigarrow^{+} \mathbf{K_1} \quad \Gamma, \alpha : K_1 \vdash K_2 : \Box \rightsquigarrow^{+} \mathbf{K_2}}{\Gamma \vdash (\Pi \alpha : K_1 . K_2) : \Box \rightsquigarrow^{+} \mathbf{\Pi \boldsymbol{\alpha} : K_1 . K_2}}$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R} \quad \Gamma, x : A \vdash K : \Box \rightsquigarrow^{+} \mathbf{K}}{\Gamma \vdash (\Pi x : A . K) : \Box \rightsquigarrow^{+} \mathbf{\Pi x : R . K}}$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{+} \mathbf{A}}{\Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{\Pi \boldsymbol{\gamma} : *_A . (A \to \boldsymbol{\gamma}) \to \boldsymbol{\gamma}}}$$

$$\frac{}{\Gamma \vdash \alpha : K \rightsquigarrow^{+} \boldsymbol{\alpha}}$$

$$\frac{\Gamma \vdash K_1 : \Box \rightsquigarrow^{+} \mathbf{K_1} \quad \Gamma, \alpha : K_1 \vdash A : K_2 \rightsquigarrow^{+} \mathbf{A}}{\Gamma \vdash (\lambda \alpha : K_1 . A) : (\Pi \alpha : K_1 . K_2) \rightsquigarrow^{+} \mathbf{\lambda \boldsymbol{\alpha} : K_1 . A}}$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R} \quad \Gamma, x : A \vdash B : K \rightsquigarrow^{+} \mathbf{B}}{\Gamma \vdash (\lambda x : A . B) : (\Pi x : A . K) \rightsquigarrow^{+} \mathbf{\lambda x : R . B}}$$

$$\frac{\Gamma \vdash A : (\Pi \alpha : K_1 . K_2) \rightsquigarrow^{+} \mathbf{A} \quad \Gamma \vdash B : K_1 \rightsquigarrow^{+} \mathbf{B}}{\Gamma \vdash A\ B : K_2\ [B/\alpha] \rightsquigarrow^{+} \mathbf{A\ B}}$$

$$\frac{\Gamma \vdash A : (\Pi x : B . K) \rightsquigarrow^{+} \mathbf{A} \quad \Gamma \vdash e : B \rightsquigarrow^{\div} \mathbf{r}}{\Gamma \vdash A\ e : K\ [e/x] \rightsquigarrow^{+} \mathbf{A\ r}}$$

$$\frac{\Gamma \vdash K_1 : \Box \rightsquigarrow^{+} \mathbf{K_1} \quad \Gamma, \alpha : K_1 \vdash A : K_2 \rightsquigarrow^{\div} \mathbf{R}}{\Gamma \vdash (\Pi \alpha : K_1 . B) : * \rightsquigarrow^{+} \mathbf{\Pi \boldsymbol{\alpha} : K_1 . R}}$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R_1} \quad \Gamma, x : A \vdash B : K \rightsquigarrow^{\div} \mathbf{R_2}}{\Gamma \vdash (\Pi x : A . B) : * \rightsquigarrow^{+} \mathbf{\Pi x : R_1 . R_2}}$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R_1} \quad \Gamma, x : A \vdash B : K \rightsquigarrow^{\div} \mathbf{R_2}}{\Gamma \vdash (\Sigma x : A . B) : * \rightsquigarrow^{+} \mathbf{\Sigma x : R_1 . R_2}}$$

$$\frac{\Gamma \vdash A : K_1 \rightsquigarrow^{+} \mathbf{A} \quad \Gamma \vdash K_1 : \Box \rightsquigarrow^{+} \mathbf{K_1} \quad \Gamma, \alpha = A : K_1 \vdash B : K_2 \rightsquigarrow^{+} \mathbf{B}}{\Gamma \vdash \text{let } \alpha = A : K_1 \text{ in } B : K_2\ [A/\alpha] \rightsquigarrow^{+} \mathbf{let\ \boldsymbol{\alpha} = A : K_1\ in\ B}}$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow^{\div} \mathbf{r} \quad \Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R} \quad \Gamma, x = e : A \vdash B : K \rightsquigarrow^{+} \mathbf{B}}{\Gamma \vdash \text{let } x = e : A \text{ in } B : K\ [e/x] \rightsquigarrow^{+} \mathbf{let\ x = r : R\ in\ B}}$$

$$\frac{\Gamma \vdash A : K_1 \rightsquigarrow^{+} \mathbf{A} \quad \Gamma \vdash K_2 : \Box \quad \Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash A : K_2 \rightsquigarrow^{+} \mathbf{A}}$$

Figure 8. CPS Translation of Universes, Kinds, and Types

$$\frac{\Gamma \vdash e : A \mid \boldsymbol{\gamma} \mid \mathbf{k} \rightsquigarrow^{\star} \mathbf{a} \quad \Gamma \vdash A : * \rightsquigarrow^{+} \mathbf{A}}{\Gamma \vdash e : A \rightsquigarrow^{\div} \mathbf{\lambda \boldsymbol{\gamma} : *_A . \lambda k : A \to \boldsymbol{\gamma} . a}}$$

$$\frac{\Gamma \vdash v : A \rightsquigarrow^{+} \mathbf{v}}{\Gamma \vdash v : A \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \boldsymbol{\kappa}\ \mathbf{v}} \qquad \frac{}{\Gamma \vdash x : A \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{x\ \boldsymbol{\omega}\ \boldsymbol{\kappa}}}$$

$$\frac{\Gamma \vdash (\Pi \alpha : K . B) : * \rightsquigarrow^{+} \mathbf{\Pi \boldsymbol{\alpha} : K . B} \quad \Gamma \vdash A : K \rightsquigarrow^{+} \mathbf{A} \quad \Gamma \vdash p : (\Pi \alpha : K . B) \mid \boldsymbol{\omega} \mid \mathbf{\lambda y : (\Pi \boldsymbol{\alpha} : K . B) . y\ A\ \boldsymbol{\omega}\ \boldsymbol{\kappa}} \rightsquigarrow^{\star} \mathbf{a}}{\Gamma \vdash p\ A : B\ [A/\alpha] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}}$$

$$\frac{\Gamma \vdash (\Pi x : A . B) : * \rightsquigarrow^{+} \mathbf{\Pi x : R . B} \quad \Gamma \vdash e : A \rightsquigarrow^{\div} \mathbf{r} \quad \Gamma \vdash p : (\Pi x : A . B) \mid \boldsymbol{\omega} \mid \mathbf{\lambda y : (\Pi x : R . B) . y\ r\ \boldsymbol{\omega}\ \boldsymbol{\kappa}} \rightsquigarrow^{\star} \mathbf{a}}{\Gamma \vdash p\ e : B\ [e/x] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}}$$

$$\frac{\Gamma \vdash v : (\Pi \alpha : K . B) \rightsquigarrow^{+} \mathbf{v} \quad \Gamma \vdash A : K \rightsquigarrow^{+} \mathbf{A}}{\Gamma \vdash v\ A : B\ [A/\alpha] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{v\ A\ \boldsymbol{\omega}\ \boldsymbol{\kappa}}}$$

$$\frac{\Gamma \vdash v : (\Pi x : A . B) \rightsquigarrow^{+} \mathbf{v} \quad \Gamma \vdash e : A \rightsquigarrow^{\div} \mathbf{r}}{\Gamma \vdash v\ e : B\ [e/x] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{v\ r\ \boldsymbol{\omega}\ \boldsymbol{\kappa}}}$$

$$\frac{\Gamma \vdash (\Sigma x : A . B) : * \rightsquigarrow^{+} \mathbf{\Sigma x : R_1 . R_2} \quad \Gamma \vdash p : (\Sigma x : A . B) \mid \boldsymbol{\omega} \mid \mathbf{\lambda y : (\Sigma x : R_1 . R_2) . fst\ y\ \boldsymbol{\omega}\ \boldsymbol{\kappa}} \rightsquigarrow^{\star} \mathbf{a}}{\Gamma \vdash \text{fst } p : A \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}}$$

$$\frac{\Gamma \vdash v : (\Sigma x : A . B) \rightsquigarrow^{+} \mathbf{v}}{\Gamma \vdash \text{fst } v : A \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{fst\ v\ \boldsymbol{\omega}\ \boldsymbol{\kappa}}}$$

$$\frac{\Gamma \vdash (\Sigma x : A . B) : * \rightsquigarrow^{+} \mathbf{\Sigma x : R_1 . R_2} \quad \Gamma \vdash p : (\Sigma x : A . B) \mid \Sigma x : A . B \mid \mathbf{id}_{\Sigma x : A . B} \rightsquigarrow^{\star} \mathbf{v}}{\Gamma \vdash \text{snd } p : B\ [\text{fst } e/x] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{let\ y = v : (\Sigma x : R_1 . R_2)\ in\ snd\ y\ \boldsymbol{\omega}\ \boldsymbol{\kappa}}}$$

$$\frac{\Gamma \vdash v : (\Sigma x : A . B) \rightsquigarrow^{+} \mathbf{v}}{\Gamma \vdash \text{snd } v : B\ [\text{fst } e/x] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{snd\ v\ \boldsymbol{\omega}\ \boldsymbol{\kappa}}}$$

$$\frac{\Gamma \vdash A : K \rightsquigarrow^{+} \mathbf{A} \quad \Gamma \vdash K : \Box \rightsquigarrow^{+} \mathbf{K} \quad \Gamma, \alpha = A : K \vdash e : B \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}}{\Gamma \vdash \text{let } \alpha = A : K \text{ in } e : B\ [A/\alpha] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{let\ \boldsymbol{\alpha} = A : K\ in\ a}}$$

$$\frac{\Gamma \vdash e_1 : A \rightsquigarrow^{\div} \mathbf{r} \quad \Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R} \quad \Gamma, x = e_1 : A \vdash e_2 : B \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}}{\Gamma \vdash \text{let } x = e_1 : A \text{ in } e_2 : B\ [e_1/x] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{let\ x = r : R\ in\ a}}$$

$$\frac{\Gamma \vdash e : A \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a} \quad \Gamma \vdash B : * \quad \Gamma \vdash A \equiv \mathbf{B}}{\Gamma \vdash e : B \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}}$$

Figure 9. Computation and Answer Translations of Terms

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow^{+} \mathbf{x}}$$

$$\frac{\Gamma \vdash K : \Box \rightsquigarrow^{+} \mathbf{K} \quad \Gamma, \alpha : K \vdash e : B \rightsquigarrow^{\div} \mathbf{r}}{\Gamma \vdash (\lambda \alpha : K.\, e) : (\Pi \alpha : K.\, B) \rightsquigarrow^{+} (\boldsymbol{\lambda \alpha} : \mathbf{K}.\, \mathbf{r})}$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{\div} \mathbf{R} \quad \Gamma, x : A \vdash e : B \rightsquigarrow^{\div} \mathbf{r}}{\Gamma \vdash (\lambda x : A.\, e) : (\Pi x : A.\, B) \rightsquigarrow^{+} (\boldsymbol{\lambda x} : \mathbf{R}.\, \mathbf{r})}$$

$$\frac{\Gamma \vdash e_1 : A \rightsquigarrow^{\div} \mathbf{r_1} \quad \Gamma \vdash e_2 : B\,[e_1/x] \rightsquigarrow^{\div} \mathbf{r_2} \quad \Gamma \vdash (\Sigma x : A.\, B) : * \rightsquigarrow^{+} \boldsymbol{\Sigma x} : \mathbf{R_1}.\, \mathbf{R_2}}{\Gamma \vdash (\langle e_1, e_2 \rangle \text{ as } \Sigma x : A.\, B) : (\Sigma x : A.\, B) \rightsquigarrow^{+} \langle \mathbf{r_1}, \mathbf{r_2} \rangle \text{ as } \boldsymbol{\Sigma x} : \mathbf{R_1}.\, \mathbf{R_2}}$$

$$\frac{\Gamma \vdash v : A \rightsquigarrow^{+} \mathbf{v} \quad \Gamma \vdash B : * \quad \Gamma \vdash A \equiv \mathbf{B}}{\Gamma \vdash v : B \rightsquigarrow^{+} \mathbf{v}}$$

Figure 10. Value Translation of Terms

have a third translation $\star$, which we call *answer translation*. It is equivalent to what is referred to as colon translation in the literature [16, 24], despite the notational difference. As the name suggests, the answer translation produces an answer (result of a computation) using a given answer type and continuation. Observe that the translation has multiple cases for most non-atomic constructs, differing in which subterms are values and which are computations. For instance, the translation has two cases for an application of two terms: when the function is a non-value p, and when the function is a value v. In the former case, the translation recursively applies itself to p with an extended continuation. In the latter case, the translation applies the value translation to v and builds an application using the result. Having both cases makes it possible to produce a compact term that is administrative redex-free.

The answer translation has one interesting case, which is the second projection of a non-value term (snd p). What is unique here is that the translation yields a `let` expression introducing the definition $\mathbf{y} = \mathbf{v}$. Here, $\mathbf{v}$ is a value (more precisely, closed answer) produced by the answer translation of p with respect to the identity continuation, which corresponds to the result of evaluating the source term p. The definition is key to establishing type preservation of the translation. Specifically, it allows us to type check the application $\mathbf{snd\ y\ \omega\ \kappa}$ with the information about what value is bound to $\mathbf{y}$. If we applied the answer translation to p while representing its continuation using a λ abstraction, we would not be able to use this information unless we augment the target language with a non-standard typing rule [10].

6 Proving Type Preservation

We are currently proving type preservation of our CPS translation. So far, we have proved representative cases while assuming several lemmas, and we are hoping to finish the proof by the time when we submit the proceedings version of this paper. Below, we provide an overview of the proof and detail a key case.

For conciseness, we use the following notations in the statements and proofs.

$$\Gamma^{+} \equiv \boldsymbol{\Gamma} \text{ where } \vdash \Gamma \rightsquigarrow^{+} \boldsymbol{\Gamma}$$
$$M^{\circ} \equiv \mathbf{M} \text{ where } \Gamma \vdash M : T \rightsquigarrow^{\circ} \mathbf{M} \text{ and } \circ \in \{\div, +\}$$
$$(e \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa})^{\star} \equiv \mathbf{a} \text{ where } \Gamma \vdash e : A \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa} \rightsquigarrow^{\star} \mathbf{a}$$

The type preservation theorem we would like to prove says: for any well-typed source term, the CPS translation produces a well-typed target term. Formally, the theorem is stated as follows.

Theorem 6.1 (Type Preservation)**.**

1. *If* $\vdash \Gamma$, *then* $\vdash \Gamma^{+}$.
2. *If* $\Gamma \vdash M : T$, *then* $\Gamma^{+} \vdash M^{\circ} : T^{\circ}$.
3. *If* $\Gamma \vdash e : A$, $\Gamma^{+} \mid \Theta \vdash \omega_\theta : *_{\mathrm{A}}$, *and* $\Gamma^{+} \mid \Theta \vdash \kappa : A^{+} \to \omega_\theta$, *then* $\Gamma^{+} \mid \Theta \vdash (e \mid \omega_\theta \mid \kappa)^{\star} : \omega_\theta$.

To prove this theorem, we need three lemmas. The first one is the substitution lemma, which states that the translation commutes with substitution.

Lemma 6.2 (Substitution)**.**

1. $(M\,[A/\alpha])^{\circ} \equiv M^{\circ}\,[A^{+}/\boldsymbol{\alpha}]$
2. $(M\,[e/x])^{\circ} \equiv M^{\circ}\,[e^{\div}/\boldsymbol{x}]$
3. $(e\,[A/\alpha] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa}\,[A^{+}/\boldsymbol{\alpha}])^{\star} \equiv (e \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa})^{\star}\,[A^{+}/\boldsymbol{\alpha}]$
4. $(e\,[e'/x] \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa}\,[e'^{\div}/\boldsymbol{x}])^{\star} \equiv (e \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa})^{\star}\,[e'^{\div}/\boldsymbol{x}]$

The second lemma is correctness, that is, the translation preserves equivalence.

Lemma 6.3 (Correctness)**.**

If $\Gamma \vdash M \equiv N$, *then* $\Gamma^{+} \vdash M^{\circ} \equiv N^{\circ}$.

The last lemma is naturality, which is a variation of Bowman et al.'s [8] [≡-Cont] taylored to our translation.

Lemma 6.4 (Naturality)**.**

For any source term $e : A$, *target type* $\boldsymbol{\omega} : *_{\mathrm{A}}$, *and target continuation* $\boldsymbol{\kappa} : A^{+} \to \boldsymbol{\omega}$, *we have* $(e \mid \boldsymbol{\omega} \mid \boldsymbol{\kappa})^{\star} \equiv \boldsymbol{\kappa}\ (e \mid A^{+} \mid \boldsymbol{id}_{A^{+}})^{\star}$.

Assuming these lemmas hold, we prove the type preservation theorem. The overall strategy is to use mutual induction on the derivation of Γ and M. Here we show one case of the third proposition (type preservation of the answer translation), where $M = \mathrm{snd}\ p$ and the last rule used is (Snd).

Our goal is to prove

$\Gamma \mid B\ [\mathsf{fst}\ p/x]^+ \vdash \mathbf{let}\ y = v : (\Sigma x : R_1.\ R_2)\ \mathbf{in}\ \mathsf{snd}\ y\ \omega\ \kappa : \omega$

where $v = (p \mid \Sigma x : A^\div.\ B^\div \mid \mathsf{id}_{\Sigma x:A^\div.\ B^\div})^\star$.

Let us first focus our attention to the typing of $\mathsf{snd}\ y$. By rule [Snd], we have

$$\Gamma, y = v : \Sigma x : A^\div.\ B^\div \vdash \mathsf{snd}\ y : B^\div\ [\mathsf{fst}\ y/x]$$

Applying δ reduction, we obtain

$$\Gamma', y = v : \Sigma x : A^\div.\ B^\div \vdash \mathsf{snd}\ y : B^\div\ [\mathsf{fst}\ v/x]$$

We next shift our attention to the domain of k. The goal says that the domain must be $(B\ [\mathsf{fst}\ p/x])^+$. By the substitution lemma, we know that it is equivalent to $B^+\ [(\mathsf{fst}\ p)^\div/x]$.

Now, we need to show

$$(\mathsf{fst}\ p)^\div \equiv \mathsf{fst}\ v$$

This equivalence can be established via the following reasoning.

$$\begin{aligned}
&(\mathsf{fst}\ p)^\div \\
&= \lambda y : *_A.\ \lambda k : A^+ \to y. \\
&\quad (p \mid y \mid \lambda y : (\Sigma x : A^\div.\ B^\div).\ (\mathsf{fst}\ y)\ y\ k)^\star \\
&\quad \text{by definition of translation} \\
&\equiv \lambda y : *_A.\ \lambda k : A^+ \to y. \\
&\quad (\lambda y : (\Sigma x : A^\div.\ B^\div).\ (\mathsf{fst}\ y)\ y\ k)\ v \\
&\quad \text{by naturality} \\
&\triangleright \lambda y : *_A.\ \lambda k : A^+ \to y.\ (\mathsf{fst}\ v)\ y\ k \\
&\quad \text{by } \beta \\
&\equiv \mathsf{fst}\ v \\
&\quad \text{by } \equiv\text{-}\eta
\end{aligned}$$

Using this equivalence and congruence, we can conclude that the goal statement holds.

It is promissing that the proof goes well for second projection, because this is the only case where the translation introduces a `let` expression. As for the lemmas used, substitution and correctness could be proved by straightforward induction on the derivation. Naturality seems to require a stronger induction principle, but it should be provable given that similar properties have been proved elsewhere [2, 31].

7 Related Work

Type-Preserving Compilation. The technique of type-preserving compilation emerged as an approach to improving safety and efficiency of generated code. Tarditi et al. [30] construct the TIL compiler for SML, where "TIL" stands for "typed intermediate languages". They show that, by performing translations and optimizations on typed intermediate languages, one can support polymorphism and garbage collection without introducing overhead, and also gain the opportunities to identify bugs in the compiler. Morrisett et al. [22] define a typed assembly language (TAL) and a type-preserving translation from System F to TAL. They demonstrate that having types in an assembly language allows one to compile higher-order functions and data types in a safe and concise manner.

Dependent-Type-Preserving Compilation. Dependent-type-preserving compilation is being studied as a way of guaranteeing safety of code compiled from languages such as Coq and Agda. Closest to our work is Bowman et al. [8], who develop call-by-name and call-by-value CPS translations of a source language that is identical to CC. As we discussed in Section 2, they equip the target language with new typing and equivalence rules to allow parametricity reasoning needed in the type preservation proof. The technique has proven applicable to CPS translations of effectful source languages as well, under the condition that no type depends on effectful terms [11, 21]. Also relevant to our work is Koronkevich et al. [20], who define an ANF translation of CoC extended with natural numbers, dependent `let`, and a universe hierarchy. They show that, unlike CPS, ANF does not require parametricity, and hence is applicable to a wider range of type theory. Other translations developed so far include defunctionalization [18], closure conversion [7], and memory allocation [19].

Non-Uniform CPS Translations. Non-uniform CPS translations like ours have mainly been developed for optimization purposes. Danvy and Hatcliff [15] define a selective CPS translation of a call-by-name language, using the result of a strictness analysis to avoid unnecessary suspensions. Nielsen [23], Rompf et al. [26], and Asai and Uehara [3] each give a selective CPS translation of a language with control operators, under the principle "convert effectful terms into CPS, keep pure terms in direct style". Reppy [25] implements a local CPS translation in the Moby compiler, which improves performance of nested loops by introducing a return continuation to non-tail calls.

Apart from these, there is a line of work that makes use of the selectiveness of CPS translations to preserve dependent types. Cong and Asai [12] and Cong [9] develop selective CPS translations of CoC extended with the delimited control operators `shift` and `reset` [14]. They establish type preservation by restricting source types to depend only on pure terms, and by keeping pure terms in direct style. This is similar to what we do; indeed, their direct style translation works like our answer translation whose continuation argument is the identity function. The difference is that their translation yields administrative redexes, and that it is no-op when applied to a pure program.

8 Conclusion and Next Steps

We present a CPS translation of a dependently typed language that produces terms involving ANF-like components.

The use of `let` expressions allows us to avoid administrative redexes while preserving dependent types, although the proof is still under development.

After proving type preservation, we will extend our results to a call-by-value CPS translation, as well as a source language with inductive data types. A call-by-value translation would involve more uses of `let` expressions, as more subterms must be evaluated for the entire term to be evaluated [8]. Inductive data types would require a target language that has extentional equality, which allows us to equate the scrutinee of a pattern matching and the pattern of a specific branch [20].

As a different direction for future work, we plan to develop a direct-style (DS) translation from CC^k to a subset of CC. Such a translation is useful for proving correctness of compilers. In particular, when a CPS translation (compiling function) and a DS translation (decompiling function) form a *reflection* [27], there is a rigorous correspondence between source and target optimizations. It would thus be beneficial to define a type-preserving DS translation and use it to establish a typed reflection.

In addition to these, we would like to investigate the usability of our translation as a compiler pass. We are especially interested in which of the known CPS and ANF optimizations are applicable to the result of the translation, and whether any new optimizations arise from the unique design of the translation. At the symposium, we hope to discuss this with other participants, especially with those who have experience in building realistic compilers.

Acknowledgments

The authors would like to thank the participants of PEPM 2024 for the inspiration that led to this paper.

References

[1] Andreas Abel. 2020. Type-preserving compilation via dependently typed syntax. Talk at 26th International Conference on Types for Proofs and Programs (TYPES '20).

[2] Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 431–444. https://doi.org/10.1145/2034773.2034830

[3] Kenichi Asai and Chihiro Uehara. 2017. Selective CPS Transformation for Shift and Reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Los Angeles, CA, USA) *(PEPM '18)*. ACM, New York, NY, USA, 40–52. https://doi.org/10.1145/3162069

[4] Gilles Barthe, John Hatcliff, and Morten Heine B Sørensen. 1999. CPS translations and applications: the cube and beyond. *Higher-Order and Symbolic Computation* 12, 2 (1999), 125–170.

[5] Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. 2020. A Reflection on Continuation-Composing Style. In *Proceedings of 5th International Conference on Formal Structures for Computation and Deduction (FSCD '20)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. 2021. *Reflecting Stacked Continuations in a Fine-Grained Direct-Style Reduction Theory*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3479394.3479399

[7] William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI '18)*. Association for Computing Machinery, New York, NY, USA, 797–811. https://doi.org/10.1145/3192366.3192372

[8] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-preserving CPS Translation of Σ and Π Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article 22 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158110

[9] Youyou Cong. 2019. *Abstracting Control with Dependent Types*. Ph. D. Dissertation. https://tinyurl.com/bdhup64f

[10] Youyou Cong. 2024. One-Pass CPS Translation of Dependent Types. Presented at the 2024 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '24). https://tinyurl.com/2ntsmy9r

[11] Youyou Cong and Kenichi Asai. 2018. Handling Delimited Continuations with Dependent Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 69 (July 2018), 31 pages. https://doi.org/10.1145/3236764

[12] Youyou Cong and Kenichi Asai. 2018. Shifting and Resetting in the Calculus of Constructions. Presented at the 19th International Symposium on Trends in Functional Programming (TFP '18). https://tinyurl.com/2p9bysrf

[13] Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

[14] Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 151–160.

[15] Olivier Danvy and John Hatcliff. 1992. CPS-transformation after strictness analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 3 (1992), 195–212.

[16] Olivier Danvy and Lasse R Nielsen. 2003. A first-order one-pass CPS transformation. *Theoretical Computer Science* 308, 1-3 (2003), 239–257.

[17] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

[18] Yulong Huang and Jeremy Yallop. 2023. Defunctionalization with Dependent Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 127 (jun 2023), 23 pages. https://doi.org/10.1145/3591241

[19] Paulette Koronkevich. 2022. Dependent-Type-Preserving Memory Allocation. Presented at POPL 2022 Student Research Competition.

[20] Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J Bowman. 2022. ANF preserves dependent types up to extensional equality. *Journal of Functional Programming* 32 (2022), e12.

[21] Étienne Miquey. 2019. A Classical Sequent Calculus with Dependent Types. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 8 (2019), 47 pages. https://doi.org/10.1145/3230625

[22] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 527–568. https://doi.org/10.1145/319301.319345

[23] Lasse R Nielsen. 2001. A selective CPS transformation. *Electronic Notes in Theoretical Computer Science* 45 (2001), 311–331.

[24] Gordon D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science* 1, 2 (1975), 125–159.

[25] John Reppy. 2002. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation* 15 (2002), 161–180.

[26] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/1596550.1596596

[27] Amr Sabry and Philip Wadler. 1997. A reflection on call-by-value. *ACM transactions on programming languages and systems (TOPLAS)* 19, 6 (1997), 916–941.

[28] Paula Severi and Erik Poll. 1994. Pure type systems with definitions. In *Logical Foundations of Computer Science: Third International Symposium (LFCS '94)*. Springer, 316–328.

[29] Guy L. Steele. 1978. *Rabbit: A Compiler for Scheme*. Technical Report. USA.

[30] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 2004. TIL: a type-directed, optimizing compiler for ML. *SIGPLAN Not.* 39, 4 (apr 2004), 554–567. https://doi.org/10.1145/989393.989449

[31] Hayo Thielecke. 2003. From Control Effects to Typed Continuation Passing. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. ACM, New York, NY, USA, 139–149. https://doi.org/10.1145/604131.604144