One-Pass CPS Translation of Dependent Types (Talk Proposal)

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

Abstract

Type preservation is an important property of compilers, especially in the case where the source and target languages are dependently typed. Our goal is to define a type-preserving, one-pass CPS translation of the Calculus of Constructions with dependent pair types. In this talk proposal, we first review the existing techniques for preserving dependent types in ordinary CPS translations, and then describe our attempt to adapt those techniques to an optimizing translation that yields no administrative redexes.

Keywords: type-preserving compilers, dependent types, CPS

1 Introduction

Type preservation is an important property of compilers that ensures safety of compiled code. In particular, when the source and target languages are both dependently typed, type preservation implies that any strong invariants captured by source types will be respected at runtime.

As a common compiler pass for functional languages, Bowman et al. [2017] study CPS translations of a dependently typed calculus. CPS translations make evaluation order explicit by introducing a continuation for every subterm, and this change in the structure of terms makes it hard to reason about equivalence of types. Bowman et al. solve this problem by allowing one to extract the value of a CPS computation and use it to establish required equivalence.

To scale their work to practical compilers, we aim to define a *one-pass* CPS translation [Danvy and Nielsen 2003] of dependent types. A one-pass CPS translation yields no administrative redexes, hence the output program is more compact and efficient. While our development is not yet complete, we have found challenges specific to one-pass translations, namely the need for a special construct for representing continuations and its impact on the structure of the translation.

In the rest of this talk proposal, we describe how existing work preserves dependent types in ordinary CPS translations, and how the ideas could be adapted to a one-pass CPS translation. For clarity, we will hereafter use a non-bold blue sans-serif font to typeset the source language of the translation, and a **bold red serif font** to typeset the target language.

2 Preserving Dependent Types in an Ordinary CPS Translation

CPS translations can be understood as sequencing computations by naming the result of each computation. In a dependently typed setting, these names may appear in types, and their appearance necessitates non-trivial reasoning of type equivalence. As a representative case, we consider the second projection of a dependent pair.

$$\mathsf{snd}\ \mathsf{e} : \mathsf{B}\,[\mathsf{fst}\ \mathsf{e}/\mathsf{x}]$$

Assuming a call-by-name semantics for the source language, we evaluate this term by first reducing e to a pair of computations $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ and then extracting the second element. To reflect this evaluation to the CPS translation, we first CPS translate e (by applying $\div$[1]) and then pass it a continuation that extracts the second element of the received value.

$$\boldsymbol{\lambda}\mathbf{k} : \mathbf{B}\,[(\mathsf{fst}\ \mathsf{e})^{\div}/\mathbf{x}] \rightarrow \bot.\ \mathsf{e}^{\div}\ (\boldsymbol{\lambda}\mathbf{y} : (\boldsymbol{\Sigma}\mathbf{x} : \mathbf{A}.\,\mathbf{B}).\ (\mathbf{snd}\ \mathbf{y})\ \mathbf{k})$$

Unfortunately, the above term would be judged ill-typed in an ordinary calculus. Specifically, the term $\mathbf{snd}\ \mathbf{y}$ expects a continuation whose domain depends on $\mathbf{fst}\ \mathbf{y}$, whereas the continuation $\mathbf{k}$ has a domain that depends on $(\mathsf{fst}\ \mathsf{e})^{\div}$.

$$\mathbf{snd}\ \mathbf{y} : (\mathbf{B}\,[\mathbf{fst}\ \mathbf{y}/\mathbf{x}] \rightarrow \bot) \rightarrow \bot \qquad \mathbf{k} : \mathbf{B}\,[(\mathsf{fst}\ \mathsf{e})^{\div}/\mathbf{x}] \rightarrow \bot$$

The good news is that it *is* possible to prove the equivalence between $\mathbf{fst}\ \mathbf{y}$ and $(\mathsf{fst}\ \mathsf{e})^{\div}$, as long as the source language has no control effects. The equivalence relies on a property called *naturality* [Thielecke 2003], which states: if the source language is pure, the continuation passed to a CPS translated term $\mathsf{e}^{\div}$ can only ever receive a value obtained by running $\mathsf{e}^{\div}$ with the identity continuation.

Using naturality, Bowman et al. [2017] define call-by-name and call-by-value CPS translations for the Calculus of Constructions [Coquand and Huet 1988] and formalize their common target language. To allow extraction of values, they translate types using a polymorphic answer type, as in $\boldsymbol{\Pi\alpha} : *.\,(\mathbf{A} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$. To specify the value passed to continuations, they equip the target language with a typing

[1]The translation is in fact defined on the derivation of terms, not on terms themselves as the notation used here would suggest. In this proposal, we choose to be slightly informal for the sake of simplicity.

PEPM '24, January 16, 2024, London, UK
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

rule for the construction $\mathbf{e}_1\ \mathbf{A}\ (\lambda \mathbf{y} : \mathbf{B}.\ \mathbf{e}_2)$ that introduces the definition $\mathbf{y} = \mathbf{e}_1\ \mathbf{B}\ \mathbf{id}$ into the typing context of $\mathbf{e}_2$. By combining these techniques, Bowman et al. were able to prove type preservation of both CPS translations.

3 Preserving Dependent Types in a One-Pass CPS Translation

The CPS translations of Bowman et al. generate a number of administrative redexes that need to be reduced in a second pass. A practical compiler usually employs an optimizing translation that converts terms and reduce redexes in one pass. The question, then, is whether we can define such a one-pass CPS translation in a type-preserving manner by reusing Bowman et al.'s techniques.

Although we do not yet have a clear answer, we identified new challenges that arise only for one-pass translations. To illustrate these challenges, we consider the one-pass translation of the second projection. Following Danvy and Nielsen [2003], we define an auxiliary colon translation to avoid administrative redexes.

$$e^{\div} = \lambda \alpha : *.\ \lambda \mathbf{k} : \mathbf{A} \rightarrow \alpha.\ e : \alpha : \mathbf{k}$$
$$\mathrm{snd}\ e : \mathbf{C} : \kappa = e : \mathbf{C} : (\lambda \mathbf{y} : (\Sigma \mathbf{x} : \mathbf{A}.\ \mathbf{B}).\ (\mathbf{snd}\ \mathbf{y})\ \kappa)$$

Notice that the colon translation carries an answer type in addition to a continuation, reflecting the fact that the CPS translation abstracts over both the answer type and the continuation.

A notable difference from an ordinary CPS translation is that the colon translation does not generate the application $e^{\div}\ \mathbf{C}\ (\lambda \mathbf{y} : (\Sigma \mathbf{x} : \mathbf{A}.\ \mathbf{B}).\ (\mathbf{snd}\ \mathbf{y})\ \mathbf{C}\ \kappa)$. This means that we cannot use Bowman et al.'s typing rule to establish well-typedness of $(\mathbf{snd}\ \mathbf{y})\ \mathbf{C}\ \kappa$.

To solve the above problem, we propose to augment the target language a new construct $\lambda(\mathbf{y} = \mathbf{e}_1 : \mathbf{A}).\ \mathbf{e}_2$, which we call a *definition lambda*. Intuitively, a definition lambda is a function that receives a specific expression. In this sense, a definition lambda works like a let expression.

With this new construct, we can perform the same reasoning as Bowman et al. do in their type preservation proof. More precisely, we represent the continuation used for the colon translation of e as a definition lambda $\lambda(\mathbf{y} = (e : (\Sigma \mathbf{x} : \mathbf{A}.\ \mathbf{B}) : \mathbf{id}) : (\Sigma \mathbf{x} : \mathbf{A}.\ \mathbf{B})).\ (\mathbf{snd}\ \mathbf{y})\ \mathbf{C}\ \kappa$, and thus be specific about what value will be substituted for the variable $\mathbf{y}$.

One question that remains unaddressed is how to type definition lambdas. Our current idea is to give them a *definition arrow type* $(\mathbf{e} : \mathbf{A}) \rightarrow \mathbf{B}$, where $\mathbf{e}$ is the expression to be passed to the function. Such a type has been used by Koronkevich et al. [2022] to preserve dependent types in an ANF translation. Since CPS and ANF translations play a similar role, it is natural to hypothesize that they require a common typing mechanism.

However, the definition arrow type poses a new challenge to the CPS translation. The source of this challenge is the fact that, for any source type A, its top-level translation $A^{\div}$ would refer to a source term e that inhabits A.

$$A^{\div} = \Pi \alpha : *.\ (((e : \mathbf{A} : \mathbf{id}) : \mathbf{A}) \rightarrow \alpha) \rightarrow \alpha \qquad \text{where } e : A$$

The dependency of the type translation on source terms seems to suggest that we need to translate terms and types in parallel. This complicates the definition of the CPS translation, which in turn complicates the induction principle of the type preservation proof.

In conclusion, we believe that it is possible to define a type-preserving, one-pass CPS translation for the Σ type. We hope to report our results in the near future.

References

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

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

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

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.

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