Toward Type-Preserving Compilation of Coq *

William J. Bowman
Northeastern University, USA
wjb@williamjbowman.com

1. Introduction

A type-preserving compiler guarantees that a well-typed source program is compiled to a well-typed target program.

Theorem 1.1 (Example Type Preservation Statement)
If $\Gamma \vdash e : t$ then $\Gamma^+ \vdash e^+ : t^+$ (where $^+$ denotes compilation)

Type-preserving compilation can support correctness guarantees about compilers, and optimizations in compiler intermediate languages (ILs). For instance, Morrisett et al. [5] use type-preserving compilation from System F to a Typed Assembly Languages (TAL) to guarantee absence of stuckness, even when linking with arbitrary (well-typed) TAL code. Tarditi et al. [8] develop a compiler for ML that uses a typed IL for optimizations.

We develop type-preserving closure conversion for the Calculus of Constructions (CC). Typed closure conversion has been studied for simply-typed languages [4, 1, 6] and polymorphic languages [4, 5]. Dependent types introduce new challenges to both typed closure conversion in particular and to type preservation proofs in general.

2. Typed Closure Conversion

We specify the correctness of closure conversion as a modified typing rule for functions that guarantees the body of the function is closed except with respect to the formal parameters:

$$\frac{\mathbf{x} : \mathbf{t}, \ldots \vdash \mathbf{e} : \mathbf{t}' \qquad \mathbf{\Gamma} \vdash \mathbf{\Pi}\,(\mathbf{x} : \mathbf{t}, \ldots).\, \mathbf{t}' : \mathbf{U}}{\mathbf{\Gamma} \vdash \boldsymbol{\lambda}\,(\mathbf{x} : \mathbf{t}, \ldots).\, \mathbf{e} : \mathbf{\Pi}\,(\mathbf{x} : \mathbf{t}, \ldots).\, \mathbf{t}'}\ [\textsc{Lam}]$$

The well-known translation is to use existential types to hide the type of the environment [4, 5, 1, 7, 6]:

$$\Pi\, \mathsf{x} : \mathsf{t}.\, \mathsf{t}' \rightsquigarrow \exists\, \boldsymbol{\alpha} : \mathbf{t}''.\, (\boldsymbol{\alpha} \times \mathbf{\Pi}\,(\mathbf{x}_{env} : \boldsymbol{\alpha}, \mathbf{x} : \mathsf{t}^+).\, \mathsf{t}'^+)$$

This avoids the problem of translating two source functions with different environments but the same type to two target functions with different types.

However, this translation fails when we have type variables, as in CC. The types t^+ and t'^+ are not closed. For example, the following incomplete derivation results from $\mathbf{A}$ being free in the types, but not in the body of the function.

$$\frac{\dfrac{\cdot \vdash \mathbf{x}_{env} : \mathbf{1}, \mathbf{x} : \mathbf{A} \text{ fails}}{\cdot, \mathbf{x}_{env} : \mathbf{1}, \mathbf{x} : \mathbf{A} \vdash \mathbf{x} : \mathbf{A}}}{\mathbf{A} : \mathbf{Set} \vdash \boldsymbol{\lambda}\,(\mathbf{x}_{env} : \mathbf{1}, \mathbf{x} : \mathbf{A}).\, \mathbf{x} : \mathbf{\Pi}\,(\mathbf{x}_{env} : \mathbf{1}, \mathbf{x} : \mathbf{A}).\, \mathbf{A}}$$

Morrisett et al. [5] simplify the polymorphic type translation of Minamide et al. [4] with the observation that, in System F, type variables are computationally irrelevant so they don't need to be included in the (run-time) environment. This does not apply when term variables can appear in types. The situation is further complicated by type-level computation; locally, we cannot decide which variables in a type are free when the type is used. Consider the following function, ascribed two different types:

$$\begin{aligned} \lambda\, \mathsf{y} : \mathsf{t}'.\, \mathsf{e} &: \mathsf{let}\, \mathsf{x} = \mathsf{e} : \mathsf{t}\, \mathsf{in}\, \Pi\, \mathsf{y} : \mathsf{t}'.\, \mathsf{x} \\ \lambda\, \mathsf{y} : \mathsf{t}'.\, \mathsf{e} &: \Pi\, \mathsf{y} : \mathsf{t}'.\, \mathsf{x} \end{aligned}$$

The variable x may or may not be free, depending on the surrounding context. We must delay the choice of which type variables to include in the closure until the type is *used*, *i.e.*, to type check the function.

We solve these problems by adapting the sophisticated translation Minamide et al. [4] give for System F to dependent types. Their translation use *translucent types*, based on translucent sums [2], to allow the types to depend on the *value* of the environment in a restricted way. They then leave the type variables free in the type translation, and close them in them in the term translation. We introduce translucent functions via the following subtyping rule:

$$\frac{\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{t}_1 \qquad \mathbf{\Gamma}, \mathbf{x}_1 = \mathbf{e} : \mathbf{t}_1, \mathbf{x}_r : \mathbf{t}_r \ldots \vdash \mathbf{t}_2 \preceq \mathbf{t}_2'}{\mathbf{\Gamma} \vdash \mathbf{\Pi}\,(\mathbf{x}_1 : \mathbf{t}_1, \mathbf{x}_r : \mathbf{t}_r \ldots).\, \mathbf{t}_2 \preceq \mathbf{e} \Rightarrow \mathbf{\Pi}\,(\mathbf{x}_r : \mathbf{t}_r \ldots).\, \mathbf{t}_2'}$$

We use the additional equivalence given by this rule to unify the free type variables in the translated type with the closed type of the translated function. The essentials of the translation are given in Figure 1.

3. Proving Type Preservation

The usual recipe for proving type preservation is: give a type translation, lift it to translate typing environments, give a term translation, then prove Theorem 1.1. In CC this recipe is "thwarted by the infernal way that everything depends on everything else" (from McBride [3]). We must simultaneously translate types, terms, and environments. Similarly, we must simultaneously prove preservation of well-typedness and well-formedness of environments. Since typing requires equivalences and subtyping, we must preserve these. Therefore, while Theorem 1.1 is our goal, we must prove the following additional lemmas.

Lemma 3.1 (Preservation of Substitution)
$(\mathsf{t}[\mathsf{t}'/\mathsf{x}])^+ \equiv \mathsf{t}^+[\mathsf{t}'^+/\mathsf{x}]$

Lemma 3.2 (Preservation of Reductions)
If $\Gamma \vdash \mathsf{e} \rhd_x \mathsf{e}'$ then $\Gamma^+ \vdash \mathsf{e}^+ \rhd^\star \mathbf{e}$ and $\mathbf{e} \equiv \mathsf{e}'^+$

Lemma 3.3 (Equivalence Preservation)
If $\Gamma \vdash \mathsf{e} \equiv \mathsf{e}'$, then $\Gamma^+ \vdash \mathsf{e}^+ \equiv \mathsf{e}'^+$

Lemma 3.4 (Preservation of Subtyping)
If $\Gamma \vdash \mathsf{t}_1 \preceq \mathsf{t}_2$, then $\Gamma^+ \vdash \mathsf{t}_1^+ \preceq \mathsf{t}_2^+$

Lemma 3.5 (Type Preservation)

1. If $\vdash \Gamma$ then $\vdash \Gamma^+$
2. If $\Gamma \vdash \mathsf{e} : \mathsf{t}$ then $\Gamma^+ \vdash \mathsf{e}^+ : \mathsf{t}^+$

Lemma 3.1 introduces crucial difficulties to our proof that this translation is type-preserving. The case for when $\mathsf{t} = \Pi\, \mathsf{x} : \mathsf{t}.\, \mathsf{t}'$

* We use a blue sans-serif font to typeset our source language and a **bold red serif font** to typeset the target. The paper will be much easier to read if viewed/printed in color.

$\boxed{\Gamma \vdash e : t \rightsquigarrow e}$ where $\Gamma \vdash e : t$

$$\frac{\Gamma \vdash t : \mathsf{U} \rightsquigarrow t \qquad \Gamma, x : t \vdash t' : \mathsf{Prop} \rightsquigarrow t'}{\Gamma \vdash \Pi\, x : t.\, t' : \mathsf{Prop} \rightsquigarrow \exists\, \alpha : \mathbf{Type}_i, x_{te} : \alpha.\, (x_{te} \Rightarrow \mathbf{\Pi}\, x : t.\, t')} \quad \ldots$$

$$\frac{\Gamma, x : t \vdash t' : \mathsf{U} \rightsquigarrow t' \qquad \Gamma, x : t \vdash e : t' \rightsquigarrow e \qquad \Gamma \vdash t : \mathsf{U} \rightsquigarrow t \qquad x_0 : t_0, \ldots, x_n : t_n = \mathrm{fv}(t, e, t') \qquad \Sigma_{env} = \mathbf{\Sigma}\, x_0 : t_0 \ldots \mathbf{\Sigma}\, x_n : t_n.\, \mathbf{1} \qquad env = \langle x_0, \ldots, x_n \rangle}{\Gamma \vdash \lambda\, x : t.\, e : \Pi\, x : t.\, t' \rightsquigarrow \mathbf{pack}\, \langle \Sigma_{env}, env, \lambda\, (x_{env} : \Sigma_{env}, x : \mathbf{let}\, x_0 = \pi_0\, x_{env} : t_0\, \mathbf{in}\, \ldots t).\rangle \; \mathbf{let}\, x_0 = \pi_0\, x_{env} : t_0\, \mathbf{in}\, \ldots e}$$

Figure 1. Closure Conversion for Terms (Excerpt)

requires that the types t and t' be left open, hence require our use of translucent types. The case where $t = \lambda\, x : t_1.\, e$ requires that we show e with t' substituted for x is equivalent to a closure whose environment contains mapping x^+ to t'^+. This requires a new equivalence rule that corresponds to η-expansion of closures.

$$\frac{\begin{array}{c}\Gamma \vdash e \rhd^\star \mathbf{pack}\, \langle \Sigma_{env}, env, \lambda\, (x_{env} : \Sigma_{env}, x : t_1).\, e_1 \rangle \\ \Gamma \vdash e' \rhd^\star e_2 \\ \Gamma, x : t_1' \vdash e_1' \equiv \mathbf{unpack}\, \langle \alpha, x_{te}, f \rangle = e_2\, \mathbf{in}\, f\, x_{te}\, x \\ \text{where } t_1' = t_1[env/x_{env}] \\ e_1' = e_1[env/x_{env}]\end{array}}{\Gamma \vdash e \equiv e'}$$

4. Future Work: Inductive Types

We have extended the translation to the Calculus of Inductive Constructions (CIC), although our type preservation proofs are not yet complete. CIC introduces two additional challenges.

The Brst challenge is to distinguish between functions, which need to be closure-converted, and other syntactic forms that are conCated with functions in CIC. For example, CIC uses the Π type to describe both functions and constants. Similarly, the typing rule for dependent case expressions heavily uses the function type to type the *motive* (a term that computes the return type of the dependent case analysis) and the branches of the case expression. If we closure-convert case naïvely, then the typing rule for ***case*** in the target language will be tied to our representation of closures. For constants, we essentially η-expand them and ensure they appear fully-applied after elaboration. After this elaboration, constants no longer have function type. We elaborate the syntax of case to directly bind arguments to the motive and branches, rather than relying on functions to bind arguments.

The second challenge is how to closure convert recursive functions and preserve the guard condition. This is particularly challenging because the guard condition is requires recursive functions to be syntactically applied to an argument guarded by a constant. However, inspection of the guard condition shows that the name of the recursive function can appear (in fact, if it is ever used then it *must* appear) in the closure of functions, such as the branches of a case expression. After closure conversion, we must allow the name of the recursive function to Cow into certain data structures, namely, the environment of closures. We develop a new Cow-sensitive guard condition to track the Cow of the name of the recursive function and ensure it is still guarded. While we do not prove that this new guard condition guarantees consistency, the proof that our closure conversion pass preserves the guard condition gives intuition as to why it should ensure consistency.

References

[1] A. Ahmed and M. Blume. Typed Closure Conversion Preserves Observational Equivalence. In *ICFP 2008*. URL `https://dl.acm.org/citation.cfm?id=1411227`.

[2] R. Harper and M. Lillibridge. A Type-theoretic Approach to Higher-order Modules with Sharing. In *POPL 1994*. URL `http://doi.acm.org/10.1145/174675.176927`.

[3] C. McBride. Outrageous but Meaningful Coincidences: Dependent Type-safe Syntax and Evaluation. In *Workshop on Generic Programming (WGP 2010)*. URL `http://doi.acm.org/10.1145/1863495.1863497`.

[4] Y. Minamide, G. Morrisett, and R. Harper. Typed Closure Conversion. In *POPL 1996*. URL `http://dx.doi.org/10.1145/237721.237791`.

[5] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *POPL 1998*. URL `http://dx.doi.org/10.1145/268946.268954`.

[6] M. S. New, W. J. Bowman, and A. Ahmed. Fully Abstract Compilation via Universal Embedding. In *ICFP 2016*. URL `http://doi.acm.org/10.1145/2951913.2951941`.

[7] J. T. Perconti and A. Ahmed. Verifying an Open Compiler Using Multi-Language Semantics. In *ESOP 2014*. URL `http://dx.doi.org/10.1007/978-3-642-54833-8_8`.

[8] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *PLDI 1996*. URL `http://doi.acm.org/10.1145/231379.231414`.