Typed Closure Conversion for the Calculus of Constructions*

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

Amal Ahmed
Northeastern University
USA
amal@ccs.neu.edu

Abstract

Dependently typed languages such as Coq are used to specify and verify the full functional correctness of source programs. Type-preserving compilation can be used to preserve these specifications and proofs of correctness through compilation into the generated target-language programs. Unfortunately, type-preserving compilation of dependent types is hard. In essence, the problem is that dependent type systems are designed around high-level compositional abstractions to decide type checking, but compilation interferes with the type-system rules for reasoning about run-time terms.

We develop a type-preserving closure-conversion translation from the Calculus of Constructions (CC) with strong dependent pairs (Σ types)—a subset of the core language of Coq—to a type-safe, dependently typed compiler intermediate language named CC-CC. The central challenge in this work is how to translate the source type-system rules for reasoning about functions into target type-system rules for reasoning about closures. To justify these rules, we prove soundness of CC-CC by giving a model in CC. In addition to type preservation, we prove correctness of separate compilation.

Keywords Dependent types, type theory, type-preserving compilation, closure conversion

ACM Reference Format:
William J. Bowman and Amal Ahmed. 2018. Typed Closure Conversion for the Calculus of Constructions. In *PLDI '18: PLDI '18: ACM SIGPLAN Conference on Programming Language Design and Implementation, June 18–22, 2018, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192372

*We use a combination of colors and fonts to distinguish different languages. Although the languages are distinguishable in black-and-white, the paper is easier to read when viewed or printed in color.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192372

1 Introduction

Full-spectrum dependently typed programming languages such as Coq have had tremendous impact on the formal verification of large-scale software. Coq has been used to specify and prove the full functional correctness the CompCert C compiler [27], the CertiKOS OS kernel [20, 21], and implementations of cryptographic primitives and protocols [6, 8]. The problem is that these proofs are about *source programs*, but we need guarantees about the *target programs*, generated by compilers, that actually end up running on machines. Projects such as CertiCoq [5], which aims to build a verified compiler for Coq in Coq, are a good first step. Unfortunately, CertiCoq throws out type information before compilation. This makes it difficult to ensure that the invariants of verified programs are respected when linking. A similar problem occurs when we extract a proven correct Coq program e to OCaml, then link with some unverified OCaml component f that violates the invariants of e and causes a segfault. Since Coq types are not preserved into OCaml, there is no way to type check f and flag that we should not link f with e. The state of the art is to tell the programmer to be careful.

Type-preserving compilation is the key to solving this problem. Types are useful for enforcing invariants in source programs, and we can similarly use them to check invariants when linking target programs. With type-preserving compilation, we could compile e and preserve its specifications into a typed target language. Then we could use type checking at link time to verify that all components match the invariants that e was originally verified against. Once we have a whole program after linking all components in a low-level typed—perhaps dependently typed—assembly language, there would no longer be a need to enforce invariants, so types could be erased to generate (untyped) machine code. Preserving *full-spectrum* dependent types has additional benefits—we could preserve proofs of full functional correctness into the generated code!

The goal in type-preserving compilation is not to develop new compiler translations, but to adapt existing translations so that they perform the same function but also preserve typing invariants. Unfortunately, these two goals are in conflict, particularly as the typing invariants become richer. The

richer the invariants the type system can express, the less freedom the compiler is permitted, and the more work required to establish typing invariants in the transformed code.

In the case of full-spectrum dependently typed languages, type-preserving compilation is hard. The essential problem is that compiler transformations disrupt the syntactic reasoning used by the type system to decide type checking. With full-spectrum dependent types, any runtime term can appear in types, so the type system includes rules for reasoning about equivalence and sometimes partially evaluating runtime terms during type checking. This works well in high level, functional languages such as the core language of Coq, but when compilers transform high-level language concepts into low-level machine concepts, we need new rules for how to reason about terms during type checking.

In the case of closure conversion, the problem is that, unlike in simply typed languages, free term variables are bound in *types* as well as terms. Intuitively, we translate a simply typed function $\Gamma \vdash \lambda x : A.e : A \to B$ into a closure $\Gamma \vdash \langle\!\langle (\lambda \Gamma, x : A.e), \mathrm{dom}(\Gamma) \rangle\!\rangle : A \to B$ where the code of the function is paired with its environment, and the code now receives its environment as an explicit argument. Note that the environment is *hidden* in the type of the closure so that two functions of the same type but with different environment still have the same type.[1] With dependent types, the *type* of a closure may refer to free variables from the environment. That is, in $\Gamma \vdash \lambda x : A.e : \Pi x : A.B$, variables from Γ can appear in A and B. After closure conversion, how can we keep the environment hidden in the type when the type must refer to the environment? That is, in the closure converted version of the above example $\Gamma \vdash \langle\!\langle (\lambda \Gamma, x : A.e), env \rangle\!\rangle : \Pi x : A.B$, how can A and B refer to env if env must remain hidden in the type?

We solve this problem for type-preserving closure conversion of the Calculus of Constructions with Σ types (CC)—a subset of the core language of Coq, and a calculus that is representative of full-spectrum dependently typed languages. Closure conversion transforms first-class functions with free variables into closures that pair closed, statically allocated code with a dynamically allocated environment containing the values of the free variables. There are two major challenges in designing new type-system rules for closures, which we discuss at a high-level in Section 3 before we formally present our results. In short, we need new type-system rules for reasoning about closures, and a way to synchronize the *type* of a closure, which depends on free variables, with the type of (closed) code, which cannot depend on free variables.

Contributions We make the following contributions:

1. We design and prove the consistency of CC-CC, a full-spectrum dependently typed compiler IL with support for statically reasoning about closures, Section 4. The proof of consistency also guarantees type safety of any programs in CC-CC—*i.e.*, linking any two components in CC-CC is guaranteed to have well-defined behavior.
2. We give a typed closure-conversion translation from CC to CC-CC Section 5.
3. Leveraging the type-preservation proof, we prove that this translation is correct with respect to separate compilation, *i.e.*, linking components in CC and then running to a value is equivalent to first compiling the components separately and then linking in CC-CC.

[1]Normally, we use existential types to hide the environment, but as we will see in Section 3, existential types cause problems with dependent types.

Universes	U	::=	$\star \mid \Box$
Expressions	e, A, B	::=	$x \mid \star \mid \mathsf{let}\ x = e : A\ \mathsf{in}\ e \mid \Pi x : A.\ B$
		$\mid$	$\lambda x : A.\ e \mid e\ e \mid \Sigma x : A.\ B$
		$\mid$	$\langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma x : A.\ B \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$
Environments	Γ	::=	$\cdot \mid \Gamma, x : A \mid \Gamma, x = e : A$

Figure 1. CC Syntax

Next, we introduce CC (Section 2), both to introduce our source language and to formally introduce dependent types, before presenting the central problem with typed closure conversion, and the main idea behind our solution (Section 3). Elided parts of figures and proofs are included in our online technical appendix [14].

2 Source: Calculus of Constructions (CC)

Our source language is a variant of the Calculus of Constructions (CC) extended with strong dependent pairs (Σ types) and η-equivalence for functions, which we typeset in a non-bold, blue, sans-serif font. This model is based on the CIC specification used in Coq [44, Chapter 4]. For brevity, we omit base types from this formal system but will freely use base types like natural numbers in examples.

We present the syntax of CC in Figure 1. Universes, or sorts, U are essentially the types of types. CC includes one impredicative universe $\star$, and one predicative universe $\Box$. Expressions have no explicit distinction between terms, types, or kinds, but we usually use the meta-variable e to evoke a term expression and A or B to evoke a type expression. Expressions include names x, the universe $\star$, functions $\lambda x : A.\ e$, application $e_1\ e_2$, dependent function types $\Pi x : A.\ B$, dependent let $\mathsf{let}\ x = e : A\ \mathsf{in}\ e'$, Σ types $\Sigma x : A.\ B$, dependent pairs $\langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma x : A.\ B$, first projections $\mathsf{fst}\ e$ and second projections $\mathsf{snd}\ e$. The universe $\Box$ is only used by the type system and is not a valid term. As syntactic sugar, we omit the type annotations on dependent let $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ and on dependent pairs $\langle e_1, e_2 \rangle$ when they are irrelevant or obvious from context. We also write function types as $A \to B$ when the result B does not depend on the argument. Environments Γ include assumptions $x : A$ that a name x has type A, and definitions $x = e : A$ that name x refers to e of type A.

We define conversion, or reduction, and definitional equivalence for CC in Figure 2. Conversion here is defined for

$\boxed{\Gamma \vdash e \rhd e'}$

$$\begin{array}{rcll} x & \rhd_\delta & e & \text{where } x = e : A \in \Gamma \\ \mathsf{let}\, x = e : A \,\mathsf{in}\, e_1 & \rhd_\zeta & e_1[e/x] & \\ (\lambda\, x : A.\, e_1)\, e_2 & \rhd_\beta & e_1[e_2/x] & \\ \mathsf{fst}\, \langle e_1, e_2 \rangle & \rhd_{\pi_1} & e_1 & \\ \mathsf{snd}\, \langle e_1, e_2 \rangle & \rhd_{\pi_2} & e_2 & \end{array}$$

$\boxed{\Gamma \vdash e \equiv e'}$

$$\frac{\Gamma \vdash e_1 \rhd^* e \qquad \Gamma \vdash e_2 \rhd^* e}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv]$$

$$\frac{\Gamma \vdash e_1 \rhd^* \lambda\, x : A.\, e \qquad \Gamma \vdash e_2 \rhd^* e_2' \qquad \Gamma, x : A \vdash e \equiv e_2'\, x}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-}\eta_1]$$

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* \lambda\, x : A.\, e \qquad \Gamma, x : A \vdash e_1'\, x \equiv e}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-}\eta_2]$$

Figure 2. CC Conversion and Equivalence

deciding equivalence between types (which include terms), but it can also be viewed as the operational semantics of CC terms. The small-step reduction $\Gamma \vdash e \rhd e'$ reduces the expression e to the term e' under the local environment Γ, which we usually leave implicit for brevity. The local environment is necessary to convert a name to its definition. Each conversion rule is labeled, and when we refer to conversion with an unlabeled arrow $e \rhd e'$, we mean that e reduces to e' by *some* reduction rule, *i.e.*, either $\rhd_\delta$, $\rhd_\zeta$, $\rhd_\beta$, $\rhd_{\pi_1}$, or $\rhd_{\pi_2}$. We write $\Gamma \vdash e \rhd^* e'$ to mean the reflexive, transitive, contextual closure of the relation $\Gamma \vdash e \rhd e'$. Essentially, $e \rhd^* e'$ runs e using the $\rhd$ relation any number of times, under any arbitrary context. We define equivalence $\Gamma \vdash e \equiv e'$ as reduction in the $\rhd^*$ relation up to η-equivalence, as in Coq [44, Chapter 4].

In Figure 3, we present the typing rules. The type system is standard.

Functions $\lambda\, x : A.\, e$ have dependent function type $\Pi\, x : A.\, B$ ([Lam]). The dependent function type describes that the function takes an argument, x, of type A, and returns something of type B where B may refer to, *i.e.*, *depends on*, the value of the argument x. We can use this to write polymorphic functions, such as the polymorphic identity function described by the type $\Pi\, A : \star.\, \Pi\, x : A.\, A$, or functions with pre/post conditions, such as the division function described by $\Pi\, x : \mathsf{Nat}.\, \Pi\, y : \mathsf{Nat}.\, \Pi\, \_ : y > 0.\, \mathsf{Nat}$, which statically ensures that we never divide by zero by requiring a proof that its second argument is greater than zero.

Applications $e_1\ e_2$ have type $B[e_2/x]$ ([App]), *i.e.*, the result type B of the function e_1 with the argument e_2 substituted for the name of the argument x. Using this rule and our example of the division function $\mathsf{div} : \Pi\, x : \mathsf{Nat}.\, \Pi\, y : \mathsf{Nat}.\, \Pi\, \_ : y > 0.\, \mathsf{Nat}$, we type check the term $\mathsf{div}\ 4\ 2 : \Pi\, \_ : 2 > 0.\, \mathsf{Nat}$. Notice that the term variable y in the type has been replaced with the value of the argument 2.

Dependent pairs $\langle e_1, e_2 \rangle$ have type $\Sigma\, x : A.\, B$ ([Pair]). Again, this type is a binding form. The type B of the second component of the pair can refer to the first component of the pair by the name x. We see in the rule [Snd] that the type of $\mathsf{snd}\, e$ is $B[\mathsf{fst}\, e/x]$, *i.e.*, the type B of the second component of the pair with the name x substituted by $\mathsf{fst}\, e$. We can use this to encode refinement types, such as the describing positive numbers by $\Sigma\, x : \mathsf{Nat}.\, x > 0$, *i.e.*, a pair of a number x with a proof that x is greater than 0.

$\boxed{\Gamma \vdash e : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star : \square}\ [\text{Ax-}^*] \qquad \frac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ [\textsc{Var}]$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x = e : A \vdash e' : B}{\Gamma \vdash \mathsf{let}\, x = e : A \,\mathsf{in}\, e' : B[e/x]}\ [\textsc{Let}]$$

$$\frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi\, x : A.\, B : \star}\ [\text{Prod-}^*] \qquad \frac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi\, x : A.\, B : \square}\ [\text{Prod-}\square]$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B}\ [\textsc{Lam}]$$

$$\frac{\Gamma \vdash e : \Pi\, x : A'.\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\, e' : B[e'/x]}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma\, x : A.\, B : \star}\ [\text{Sig-}^*]$$

$$\frac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Sigma\, x : A.\, B : \square}\ [\text{Sig-}\square] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{fst}\, e : A}\ [\textsc{Fst}]$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x]}\ [\textsc{Snd}]$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : B}\ [\textsc{Conv}]$$

Figure 3. CC Typing

Since types are also terms, we have typing rules for types. The type of $\star$ is $\square$. We call $\star$ the universe of small types and $\square$ the universe of large types. Intuitively, small types are the types of programs while large types are the types of types and type-level computations. Since no user can write down $\square$, we need not worry about the type of $\square$. In [Prod-*], we assign the type $\star$ to the dependent function type when the result type is also $\star$. This rule allows *impredicative* functions, since it allows forming a function that quantifies over large types but is in the universe of small types. The rule [Prod-$\square$] looks similar, but is implicitly predicative, since there is no universe larger than $\square$ to quantify over. (We could combine the rules for Π, but explicit separation helps clarify the issue of predicativity when compared with the rules for Σ types, which cannot be combined.) Formation rules for Σ types have an important restriction: it is unsound to allow impredicativity in strong dependent pairs [17, 23]. The [Sig-*] rule only allows quantifying over a small type when forming a small dependent pair. The [Sig-$\square$] rule allows quantifying

$$\boxed{\vdash \Gamma}$$

$$\frac{}{\vdash \cdot}\ [\text{W-Empty}] \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : U}{\vdash \Gamma, x : A}\ [\text{W-Assum}]$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash e : A \qquad \Gamma \vdash A : U}{\vdash \Gamma, x = e : A}\ [\text{W-Def}]$$

Figure 4. CC Well-Formed Environments

over either small or large types when forming a large Σ. As usual in models of dependent type theory, we exclude base types, although they are simple to add.

The rule [Conv] allows resolving type equivalence and reducing terms in types. For instance, if we want to show that $e : \Sigma\, x : \text{Nat}.\, x = 2$ but we have $e : \Sigma\, x : \text{Nat}.\, x = 1 + 1$, the [Conv] rule performs this reduction. Note while our equivalence relation is untyped, the [Conv] rule ensures that A and B are well-typed before appealing to equivalence, ensuring decidability. (It is a standard lemma that if $\Gamma \vdash e : A$, then $\Gamma \vdash A : U$ [28].)

Finally, we extend well-typedness to well-formedness of environments $\vdash \Gamma$ in Figure 4.

3 Main Ideas

Closure conversion makes the implicit closures from a functional language explicit to facilitate statically allocating functions in memory. The idea is to translate each first-class function into an explicit closure, *i.e.*, a pair of closed *code* and an environment data structure containing the values of the free variables. We use *code* to refer to functions with no free variables, as in a closure-converted language. The environment is created dynamically, but the closed code can be lifted to the top-level and statically allocated. Consider the following example translation.

$$\begin{aligned}(\lambda x.y)^+ &= \langle (\lambda n\, x.\, \text{let } y = (\pi_1\ n) \text{ in } y), \langle y \rangle \rangle \\ ((\lambda x.y)\ \text{true})^+ &= \text{let } \langle f, n \rangle = \langle (\lambda n\, x.\, \text{let } y = (\pi_1\ n) \text{ in } y), \langle y \rangle \rangle \text{ in} \\ &\quad f\ n\ \text{true}\end{aligned}$$

We write e^+ to indicate the translation of an expression e. We translate each function into a pair of code and its environment. The code accepts its free variables in an environment argument, n (since n sounds similar to *env*). In the body of the code, we bind the names of all free variables by projecting from this environment n. To call a closure, we apply the code to its environment and its argument.

This translation is not type preserving since the structure of the environment shows up in the type. For example, the following two functions have the same type in the source, but end up with different types in the target.

$$\begin{aligned}(\lambda x.y)^+ &: ((\text{Nat} \times \text{Nil}) \to \text{Nat} \to \text{Nat}) \times (\text{Nat} \times \text{Nil}) \\ (\lambda x.x)^+ &: (\text{Nil} \to \text{Nat} \to \text{Nat}) \times \text{Nil}\end{aligned}$$

This is a well-known problem with typed closure conversion, so we could try the well-known solution [30, 34, 35, 2, 41, 37]. (Spoiler alert: it won't work for CC.) We represent closures as an existential package of a pair of the function and its environment, whose type is hidden. The existential type hides the structure of the environment in the type.

$$\begin{aligned}(\lambda x.y)^+ &: \exists \alpha.(\alpha \to \text{Nat} \to \text{Nat}) \times \alpha \\ (\lambda x.x)^+ &: \exists \alpha.(\alpha \to \text{Nat} \to \text{Nat}) \times \alpha\end{aligned}$$

This works well for simply typed and polymorphic languages, but when we move to a dependently typed language, we have new challenges. First, the environment must now be ordered since the type of each new variable can depend on all prior variables. Second, types can now refer to variables in the closure's environment. Recall the polymorphic identity function from earlier.

$$\lambda\, A : \star.\, \lambda\, x : A.\, x \quad : \quad \Pi\, A : \star.\, \Pi\, x : A.\, A$$

This function takes a type variable, A, whose type is $\star$. It returns a function that accepts an argument x of type A and returns it. There are two closures in this example: the outer closure has no free variables, and thus will have an empty environment, while the inner closure $\lambda\, x : A.\, x$ has A free, and thus A will appear in its environment.

Below, we present the translation of this example using the previous translation. We typeset target language terms produced by our translation in a **bold, red, serif font**. We produce two closures, one nested in the other. Note that we translate source variables x to $\mathbf{x}$. In the outer closure, the environment is empty $\langle\rangle$, and the code simply returns the inner closure. The inner closure has the argument $\mathbf{A}$ from the outer code in its environment. Since the inner code takes an argument of type $\mathbf{A}$, we project $\mathbf{A}$ from the environment *in the type annotation* for $\mathbf{x}$. That is, the inner code takes an environment $\mathbf{n_2}$ that contains $\mathbf{A}$, and the type annotation for $\mathbf{x}$ is $\mathbf{x} : \mathbf{fst}\ \mathbf{n_2}$. The type $\mathbf{fst}\ \mathbf{n_2}$ is unusual, but is no problem since dependent types allow computations in types.

$$\begin{aligned}&\langle\!\langle \mathbf{\lambda}\, (\mathbf{n_1} : \mathbf{1}, \mathbf{A} : \star).\ \langle\!\langle \mathbf{\lambda}\, (\mathbf{n_2} : \star \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\ \mathbf{n_2}).\ \mathbf{x}, \langle \mathbf{A}, \langle\rangle \rangle \rangle\!\rangle, \langle\rangle \rangle\!\rangle\ : \\ &\exists\, \boldsymbol{\alpha_1} : \star.\, (\mathbf{\Pi}\, (\mathbf{n_1} : \boldsymbol{\alpha_1}, \mathbf{A} : \star). \\ &\qquad \exists\, \boldsymbol{\alpha_2} : \square.\, (\mathbf{\Pi}\, (\mathbf{n_2} : \boldsymbol{\alpha_2}, \mathbf{x} : \mathbf{fst}\ \mathbf{n_2}).\ \mathbf{fst}\ \mathbf{n_2}) \times \boldsymbol{\alpha_2}) \times \boldsymbol{\alpha_1}\end{aligned}$$

We see that the inner code on its own is well typed with the closed type $\mathbf{\Pi}\, (\mathbf{n_2} : \star \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\ \mathbf{n_2}).\ \mathbf{fst}\ \mathbf{n_2}$. That is, the code takes two arguments: the first argument $\mathbf{n_2}$ is the environment, and the second argument $\mathbf{x}$ is a value of type $\mathbf{fst}\ \mathbf{n_2}$. The result type of the code is also $\mathbf{fst}\ \mathbf{n_2}$. As discussed above, we must hide the type of the environment to ensure type preservation. That is, when we build the closure $\langle\!\langle \mathbf{\lambda}\, (\mathbf{n_2} : \star \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\ \mathbf{n_2}).\ \mathbf{x}, \langle \mathbf{A}, \langle\rangle \rangle \rangle\!\rangle$, we must hide the type of the environment $\langle \mathbf{A}, \langle\rangle \rangle$. We use an existential type to quantify over the type $\boldsymbol{\alpha_2}$ of the environment, and we produce the type $\mathbf{\Pi}\, (\mathbf{n_2} : \boldsymbol{\alpha_2}, \mathbf{x} : \mathbf{fst}\ \mathbf{n_2}).\ \mathbf{fst}\ \mathbf{n_2}$ for the code in the inner closure. But this type is trying to take the first projection of something of type $\boldsymbol{\alpha_2}$. We can only project from pairs, and something of type $\boldsymbol{\alpha_2}$ isn't a pair! In hiding the type of the environment to recover type preservation, we've broken type preservation for dependent types.

A similar problem also arises when closure converting System F, since System F also features type variables [30, 35]. To understand our solution, it is important to understand

why the solutions that have historically worked for System F do not scale to CC. We briefly present these past results and why they do not scale before moving on to the key idea behind our translation. Essentially, past work using existential types relies on assumptions about computational relevance, parametricity, and impredicativity that do not necessarily hold in full-spectrum dependent type systems.

3.1 Why the Well Known Solution Doesn't Work

Minamide et al. [30] give a translation that encodes closure types using existential types, a standard type-theoretic feature that they use to make environment hiding explicit in the types. In essence, they encode closures as objects; the environment can be thought of as the private field of an object. Since then, the use of existential types to encode closure types has been standard in all work on typed closure conversion.

However, the use of existential types to encode closures in a dependently typed setting is problematic. First, let us just consider closure conversion for System F. As Minamide et al. [30] observed, there is a problem when code must be closed with respect to both term and *type* variables. This problem is similar to the one discussed above: when closure environments contain type variables, since those type variables can also appear in the closure's type, the closure's type needs to project from the closure's (hidden) environment which has type α. To fix the problem, they extend their target language with *translucency* (essentially, a kind of type-level equivalence that we now call singleton types), type-level pairs, and kinds. All of these features can be encoded in CC, so we could extend their translation essentially as follows.

$$(\Pi\, x : A.\, B)^+ \stackrel{\text{def}}{=} \exists\, \alpha : U.\, \exists\, n : \alpha.\, \mathsf{Code}\,(n' : \alpha, y : n' = n, x : A^+).\, B^+$$

In this translation, we would existentially quantify over the type of the environment α, the *value* of the environment n, and generate code that requires an environment n' plus a proof that the code is only ever given the environment n as the argument n'. The typing rule for an existential package copies the existential value into the type. That is, for a closure $\mathsf{pack}\,\langle A', v, e\rangle$ of type $\exists\, \alpha : U.\, \exists\, n : \alpha.\, \mathsf{Code}\,(n' : \alpha, y : n' = n, x : A^+).\, B^+$, the typing rule for pack requires that we show $e : \mathsf{Code}\,(n' : A', y : n' = v, x : A^+).\, B^+$; notice that the variable n has been replaced by the value of the environment v. The equality $n' = v$ essentially unifies projections from n' with projections from v, the list of free variables representing the actual environment.

The problem with this translation is that it relies on *impredicativity*. That is, if $(\Pi\, x : A.\, B) : \star$, then we require that $(\Pi\, x{:}A.\, B)^+ : \star$. Since the existential type quantifies over a type in an arbitrary universe U but must be in the base universe $\star$, the existential type must be impredicative. Impredicative existential types (weak dependent sums) are consistent on their own, but impredicativity causes inconsistency when combined with other features, including computational relevance and Coq's universe hierarchy. In Coq by default, the base computationally relevant universe `Set` is predicative, so this translation would not work. There is a flag to enable impredicative `Set`, but this can introduce inconsistency with some axioms, such as a combination of the law of excluded middle plus the axiom of choice, or ad-hoc polymorphism [12]. Even with impredicative `Set`, there are computationally relevant universes higher in Coq's universe hierarchy, and it would not be safe to allow impredicativity at more than one universe. Furthermore, some dependently typed languages, such as Agda, do not allow impredicativity at all since it is the source of paradoxes, such as Girard's paradox.

A second problem arises in developing an η principle, because the existential type encoding relies on *parametricity* to hide the environment. So, any η principle would need to be justified by a parametric relation on environments. Internalizing parametricity for dependent type theory is an active area of research [11, 25, 26, 38] and not all dependent type theories admit parametricity [12].

Later, Morrisett et al. [35] improved the existential-type translation for System F, avoiding translucency and kinds by relying on *type erasure* before runtime, which meant that their code didn't have to close over type variables. This translation does not apply in a dependently typed setting, since now types can contain term variables not just "type erasable" type variables.

3.2 Our Translation

To solve type-preserving closure conversion for CC, we avoid existential types altogether and instead take inspiration from the so-called "abstract closure conversion" of Minamide et al. [30]. They add new forms to the target language to represent code and closures for a simply typed source language. We scale the design of these forms to dependent types.

Adapting and scaling even a well-known translation to dependent type theory is complex. Recall from Section 1 that the goal of our compiler is to implement the same functionality as standard closure conversion, but preserve the typing invariants. Operationally, our translation will do the obvious thing, but the complexity of our translation comes from the types. In the case of dependent types, the complexity (and usefulness) of the type system comes from the ability to interpret terms as logical formulas that are capable of expressing mathematical theorems and proofs. When we add new typing rules to the target language, we must justify that the new system is still consistent when interpreted as a logic. Moreover, we must design new equivalence rules for terms and, ideally, ensure that equivalence is still decidable.

In the case of closure conversion, we are transforming the fundamental feature of dependent type theory: functions and Π types. Functions can be interpreted as proofs of universal properties represented by Π types. This transformation requires dependent types for both code and closures, and a novel equivalence principle for closures. But in proving

the new rules consistent, we must not just prove that we do not allow proofs of False in the new system, but also establish that all universal properties and their proofs that were representable and provable in the source language are still representable and provable in the target language. We leave the proofs of these properties until Section 4.1, but present the key typing and equivalence rules now.

We extend our type system with primitive types for code and closures. We represent code as $\lambda\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{e}_1$ of the *code type* $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{B}$. These are still dependent types, so $\mathbf{n}$ may appear in both $\mathbf{A}$ and $\mathbf{B}$, and $\mathbf{x}$ may appear in $\mathbf{B}$. Code must be well typed in an empty environment, *i.e.*, when it is closed. For simplicity, code only takes two arguments.

$$\frac{\cdot, \mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A} \vdash \mathbf{e} : \mathbf{B}}{\Gamma \vdash \lambda\,\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}.\ \mathbf{e} : \mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{B}}\ [\textsc{Code}]$$

We represent closures as $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ of type $\Pi\,\mathbf{x}{:}\mathbf{A}[\mathbf{e}'/\mathbf{n}].\ \mathbf{B}[\mathbf{e}'/\mathbf{n}]$, where $\mathbf{e}$ is code and $\mathbf{e}'$ is its environment. We continue to use Π types to describe closures; note that "functions" in CC are implicit closures. The typing rule for closures is:

$$\frac{\Gamma \vdash \mathbf{e} : \mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{B} \qquad \Gamma \vdash \mathbf{e}' : \mathbf{A}'}{\Gamma \vdash \langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle : \Pi\,\mathbf{x} : \mathbf{A}[\mathbf{e}'/\mathbf{n}].\ \mathbf{B}[\mathbf{e}'/\mathbf{n}]}\ [\textsc{Clo}]$$

We should think of a closure $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ not as a pair, but as a delayed partial application of the code $\mathbf{e}$ to its environment $\mathbf{e}'$. This intuition is formalized in the typing rule since the environment is substituted into the type, just as in dependent-function application in CC.

To understand our translation, let us start with the translation of functions.

$$(\lambda\,\mathbf{x} : \mathbf{A}.\ \mathbf{e})^+ \stackrel{\text{def}}{=} \langle\!\langle (\lambda\,(\mathbf{n} : \Sigma\,(\mathbf{x}_i : \mathbf{A}_i^+ \ldots),\ \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{A}^+).\ \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e}^+), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle$$
where $\mathbf{x}_i : \mathbf{A}_i \ldots$ are the free variables of $\mathbf{e}$ and $\mathbf{A}$

The translation of functions is simple to construct. We know we want to produce a closure containing code and its environment. We know the environment should be constructed from the free variables of the body of the function, namely $\mathbf{e}$, and, due to dependent types, the type annotation $\mathbf{A}$.

The question is: what should the type translation of Π types be? Let's return to our polymorphic identity function (just the inner closure). If we apply the above translation, we produce the following for the inner closure. We know its type by following the typing rules [Clo] and [Code] above.

$$\langle\!\langle \lambda\,(\mathbf{n}_2 : \star \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\,\mathbf{n}_2).\ \mathbf{x}, \langle \mathbf{A}, \langle\rangle \rangle \rangle\!\rangle :$$
$$\Pi\,(\mathbf{x} : (\mathbf{fst}\,\mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle \rangle / \mathbf{n}_2]).\ (\mathbf{fst}\,\mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle \rangle / \mathbf{n}_2]$$

We know that the code $\lambda\,(\mathbf{n}_2 : \star \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\,\mathbf{n}_2).\ \mathbf{x}$ has type $\mathbf{Code}\,(\star \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\,\mathbf{n}_2).\ \mathbf{fst}\,\mathbf{n}_2$. Following [Clo], we substitute the environment into this type, so we get:

$$\Pi\,(\mathbf{x} : (\mathbf{fst}\,\mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle \rangle / \mathbf{n}_2]).\ (\mathbf{fst}\,\mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle \rangle / \mathbf{n}_2]$$

So how do we translate the function type $\Pi\,\mathbf{x}{:}\mathbf{A}.\ \mathbf{A}$ into the closure type $\Pi\,(\mathbf{x} : (\mathbf{fst}\,\mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle \rangle / \mathbf{n}_2]).\ (\mathbf{fst}\,\mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle \rangle / \mathbf{n}_2]$? Note that this type reduces to $\Pi\,\mathbf{x} : \mathbf{A}.\ \mathbf{A}$. So by the rule [Conv], we simply need to translate $\Pi\,\mathbf{x} : \mathbf{A}.\ \mathbf{A}$ to $\Pi\,\mathbf{x} : \mathbf{A}.\ \mathbf{A}$!

$$\textit{Expressions}\ \ \mathbf{e}, \mathbf{A}, \mathbf{B} ::= \cdots \mid \mathbf{1} \mid \langle\rangle \mid \mathbf{Code}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{B} \mid \lambda\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{e} \mid \Pi\,\mathbf{x} : \mathbf{A}.\ \mathbf{B} \mid \langle\!\langle \mathbf{e}, \mathbf{e} \rangle\!\rangle$$

Figure 5. CC-CC Syntax (excerpts)

The key translation rules are given below.

$$(\Pi\,\mathbf{x} : \mathbf{A}.\ \mathbf{B})^+ \stackrel{\text{def}}{=} \Pi\,\mathbf{x} : \mathbf{A}^+.\ \mathbf{B}^+$$
$$(\lambda\,\mathbf{x} : \mathbf{A}.\ \mathbf{e})^+ \stackrel{\text{def}}{=} \langle\!\langle (\lambda\,(\mathbf{n} : \Sigma\,(\mathbf{x}_i : \mathbf{A}_i^+ \ldots),\ \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{A}^+).\ \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e}^+), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle$$
where $\mathbf{x}_i : \mathbf{A}_i \ldots$ are the free variables of $\mathbf{e}$ and $\mathbf{A}$

A final challenge remains in the design of our target language: we need to know when two closures are equivalent. As we just saw, CC partially evaluates terms while type checking. If two closures get evaluated while resolving type equivalence, we may inline a term into the environment for one closure but not the other. When this happens, two closures that were syntactically identical and thus equivalent become inequivalent. We discuss this problem in detail in Section 5, but essentially we need to know when two syntactically distinct closures are equivalent. Our solution is simple: get rid of the closures and keep inlining things!

$$\frac{\Gamma, \mathbf{x} : \mathbf{A} \vdash \mathbf{e}_1[\mathbf{e}_1'/\mathbf{n}] \equiv \mathbf{e}_2[\mathbf{e}_2'/\mathbf{n}]}{\Gamma \vdash \langle\!\langle (\lambda\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{e}_1), \mathbf{e}_1' \rangle\!\rangle \equiv \langle\!\langle (\lambda\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{e}_2), \mathbf{e}_2' \rangle\!\rangle}$$

Two closures are equivalent when we inline the environment, free variables or not, and run the body of the code. We leave the argument free, too. We run the bodies of the code to normal forms, then compare the normal forms. Recall that equivalence runs terms while *type checking* and does not change the program, so the free variables do no harm.

This equivalence essentially corresponds to an η-principle for closures. From it, we can derive a normal form for closures $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ that says the environment $\mathbf{e}'$ contains only free variables, *i.e.*, $\mathbf{e}' = \langle \mathbf{x}_i \ldots \rangle$.

The above is an intuitive, declarative presentation, but is incomplete without additional rules. We use an algorithmic presentation that is similar to the η-equivalence rules for functions in CC, which we show in Section 4.

4 Target: CC, Closure-Converted (CC-CC)

The target language CC-CC is based on CC, but first-class functions are replaced by closed code and closures. We add a primitive unit type $\mathbf{1}$ to support encoding environments. We extend the syntax of expressions, Figure 5, with a unit value $\langle\rangle$ and its type $\mathbf{1}$, closed code $\lambda\,\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}.\ \mathbf{e}$ and dependent code types $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{B}$, and closure values $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ and dependent closure types $\Pi\,\mathbf{x} : \mathbf{A}.\ \mathbf{B}$. The syntax of application $\mathbf{e}\ \mathbf{e}'$ is unchanged, but it now applies closures instead of functions.

We define additional syntactic sugar for sequences of terms, to support writing environments whose length is arbitrary. We write a sequence of terms $\mathbf{e}_i \ldots$ to mean a sequence of length $|i|$ of expressions $\mathbf{e}_{i_0}, \ldots, \mathbf{e}_{i_n}$. We extend the notation to patterns such as $\mathbf{x}_i : \mathbf{A}_i \ldots$, which implies

$\boxed{\Gamma \vdash e \rhd e'}$

$$\vdots$$

$$\langle\!\langle \lambda\, x' : A', x : A.\, e_1, e' \rangle\!\rangle\ e \quad \rhd_\beta \quad e_1[e'/x'][e/x]$$

$\boxed{\Gamma \vdash e \equiv e'}$

$$\cdots \qquad \frac{\Gamma \vdash e_1 \rhd^* \langle\!\langle \lambda\,(x' : A', x : A).\, e_1', e' \rangle\!\rangle \quad \Gamma \vdash e_2 \rhd^* e_2' \qquad \Gamma, x : A \vdash e_1[e'/x'] \equiv e_2'\ x}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Clo}_1]$$

$$\frac{\Gamma \vdash e_2 \rhd^* \langle\!\langle \lambda\,(x' : A', x : A).\, e_2', e' \rangle\!\rangle \quad \Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x : A \vdash e_1'\ x \equiv e_2'[e'/x']}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Clo}_2]$$

Figure 6. CC-CC Conversion and Equivalence (excerpts)

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

$$\cdots \qquad \frac{\Gamma, x' : A', x : A \vdash B : \star}{\Gamma \vdash \mathbf{Code}\,(x' : A', x : A).\, B : \star}\ [\text{T-Code-}*]$$

$$\frac{\Gamma, x' : A', x : A \vdash B : \Box}{\Gamma \vdash \mathbf{Code}\,(x : A, x' : A').\, B : \Box}\ [\text{T-Code-}\Box]$$

$$\frac{\cdot, x' : A', x : A \vdash e : B}{\Gamma \vdash \lambda\,(x' : A', x : A).\, e : \mathbf{Code}\,(x' : A', x : A).\, B}\ [\text{Code}]$$

$$\frac{\Gamma \vdash e : \mathbf{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash \langle\!\langle e, e' \rangle\!\rangle : \Pi\, x : A[e'/x'].\, B[e'/x']}\ [\text{Clo}]$$

Figure 7. CC-CC Typing (excerpts)

two sequences $x_{i_0}, \ldots, x_{i_n}$ and $A_0, \ldots, A_{i_n}$ each of length $|i|$. We define environments as dependent n-tuples, written $\langle e_i \ldots \rangle\ \mathbf{as}\ \Sigma\,(x_i : A_i \ldots)$. We encode dependent n-tuples as nested dependent pairs followed by a unit value, *i.e.*, $\langle e_0, \langle \ldots, \langle e_i, \langle\rangle \rangle \rangle \rangle$. We omit the annotation on n-tuples $\langle e_i \ldots \rangle$ when it is obvious from context. We also define pattern matching on n-tuples, written $\mathbf{let}\ \langle x_i \ldots \rangle = e'\ \mathbf{in}\ e$, to perform the necessary nested projections, *i.e.*, $\mathbf{let}\ x_0 = \mathbf{fst}\ e'\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_i = \mathbf{fst}\ \mathbf{snd}\ \ldots\ \mathbf{snd}\ e'\ \mathbf{in}\ e$.

In Figure 6 we present the additional conversion and equivalence rules for CC-CC. Code cannot be applied directly, but must be part of a closure. Closures applied to an argument β-reduce, applying the underlying code to the environment and the argument. All the other conversion rules remain unchanged. For equivalence, we no longer have the usual η rules, since functions have been turned into closures. Instead, we need η rules for closures.

We give the typing rules in Figure 7. All unspecified rules are unchanged from the source language. The most interesting rule is [Code], which that code only type checks when it is closed. This rule captures the entire point of typed closure conversion and gives us static machine-checked guarantees that our translation produces closed code. The typing rule [Clo] for closures $\langle\!\langle e, e' \rangle\!\rangle$ substitutes the environment e' into the type of the closure, as discussed in Section 3. This is similar to the CC rule [App] that substitutes a function argument into the result type of a function. As we discussed in Section 3, this is also critical to type preservation, since our translation must generate closure types with free variables and then synchronize the closure type containing free variables with a closed code type. As with Π types in CC, we have two rules for well typed Code types. The rule [T-Code-*] allows impredicativity in $\star$, while [T-Code-$\Box$] is predicative.

4.1 Type Safety and Consistency

We prove that CC-CC is type safe when interpreted as a programming language and consistent when interpreted as a logic. Type safety guarantees that all programs in CC-CC have well-defined behavior, and consistency ensures that when interpreting types as propositions and programs as proofs, we cannot prove False in CC-CC. We prove both theorems by giving a model of CC-CC in CC, *i.e.*, by encoding the target language in the source language. The model reduces type safety and consistency of CC-CC to that of CC, which is known to be type safe and consistent. This standard technique is well explained by Boulier et al. [12].

We construct a model essentially by "decompiling" closures, translating code to functions and closures to partial application. To show this translation is a model, we need to show that it preserves falseness—*i.e.*, that we translate False to False—and show that the translation is type-preserving—*i.e.*, we translate any well-typed CC-CC program (valid proof) into a well-typed program in CC. To extend the model to type safety, we must also show that the translation preserves reduction semantics—*i.e.*, that reducing an expression in CC-CC is essentially equivalent to reducing the translated term in CC. Since our type system includes reduction, we already prove this to show type preservation.

We then prove consistency and type safety of CC-CC by contradiction. If CC-CC were inconsistent, then we could prove the proposition False in CC-CC, and translate that proof into a valid proof of False in CC. But since CC is consistent, we can never produce a proof of False in CC, therefore we could not have constructed one in CC-CC. A similar argument applies for type safety. Since we preserve reduction semantics in CC-CC, if a term had undefined behavior, we could translate the term into a CC term with undefined behavior. However, CC has no terms with undefined behavior, hence neither does CC-CC.

The translation from CC-CC to CC, Figure 8, is defined on typing derivations. We use the following notation.

$$e^\circ \stackrel{\text{def}}{=} e \text{ where } \Gamma \vdash e : A \rightsquigarrow_\circ e$$

The CC expression e° refers to the expression produced by translating the CC-CC expression e, with the typing derivation for e as an implicit argument.

The rule [M-Code] translates a code type $\mathbf{Code}\,(n : A', x : A).\, B$ to the curried function type $\Pi\, n{:}A'^\circ.\, \Pi\, x{:}A^\circ.\, B^\circ$. The rule [M-Code] models code $\lambda\, n : A', x : A.\, e$ as a curried function

$\boxed{\Gamma \vdash e : A \rightsquigarrow_\circ e}$

$$\cdots \qquad \frac{\Gamma \vdash A : U \rightsquigarrow_\circ A \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow_\circ B}{\Gamma \vdash \Pi\, x : A.\, B : \star \rightsquigarrow_\circ \Pi\, x : A.\, B}\ [\text{M-Prod-}*]$$

$$\frac{\Gamma \vdash A' : U' \rightsquigarrow_\circ A' \qquad \Gamma, x' : A' \vdash A : U \rightsquigarrow_\circ A \qquad \Gamma, x' : A', x : A \vdash B : \star \rightsquigarrow_\circ B}{\Gamma \vdash \mathbf{Code}\,(x' : A', x : A).\, B : \star \rightsquigarrow_\circ \Pi\, x' : A'.\, \Pi\, x : A.\, B}\ [\text{M-T-Code-}*]$$

$$\frac{\Gamma \vdash A' : U' \rightsquigarrow_\circ A' \qquad \Gamma, x' : A' \vdash A : U \rightsquigarrow_\circ A \qquad \Gamma, x' : A', x : A \vdash B : \square \rightsquigarrow_\circ B}{\Gamma \vdash \mathbf{Code}\,(x' : A', x : A).\, B : \square \rightsquigarrow_\circ \Pi\, x' : A'.\, \Pi\, x : A.\, B}\ [\text{M-T-Code-}\square]$$

$$\frac{\Gamma \vdash A' : U' \rightsquigarrow_\circ A' \qquad \Gamma, x' : A' \vdash A : U \rightsquigarrow_\circ A \qquad \Gamma, x' : A', x : A \vdash B : U \rightsquigarrow_\circ B \qquad \Gamma, x' : A', x : A \vdash e : B \rightsquigarrow_\circ e}{\Gamma \vdash \lambda\,(x' : A', x : A).\, e : \mathbf{Code}\,(x' : A', x : A).\, B \rightsquigarrow_\circ \lambda\, x' : A'.\, \lambda\, x : A.\, e}\ [\text{M-Code}]$$

$$\frac{\Gamma \vdash e : \mathbf{Code}\,(x' : A', x : A).\, B \rightsquigarrow_\circ e \qquad \Gamma \vdash e' : A' \rightsquigarrow_\circ e'}{\Gamma \vdash \langle\!\langle e, e' \rangle\!\rangle : \Pi\, x : A[e'/x].\, B[e'/x] \rightsquigarrow_\circ e\ e'}\ [\text{M-Clo}] \qquad \frac{\Gamma \vdash e : \Pi\, x : A.\, B \rightsquigarrow_\circ e \qquad \Gamma \vdash e' : A \rightsquigarrow_\circ e'}{\Gamma \vdash e\ e' : B[e'/x] \rightsquigarrow_\circ e\ e'}\ [\text{M-App}]$$

Figure 8. Translation from CC-CC to CC (excerpts)

$\lambda\, n : A'^\circ.\, \lambda\, x : A^\circ.\, e^\circ$. Observe that the inner function produced in CC is not closed, but that is not a problem since the model only exists to prove type safety and consistency. It is only in CC-CC programs that code must be closed. The rule [M-Clo] models a closure $\langle\!\langle e, e' \rangle\!\rangle$ as the application $e^\circ\ e'^\circ$—*i.e.*, the application of the function e° to its environment e'°. We model Unit, omitted for brevity, with the standard Church encoding as the polymorphic identity function. All other rules simply recursively translate subterms.

We first prove that this translation preserves falseness. We encode False in CC-CC as $\Pi\, A : \star.\, A$. This encoding represents a function that takes any arbitrary proposition A and returns a proof of A. Similar, in CC False as $\Pi\, A : \star.\, A$. It is clear from [M-Prod-*] that the translation preserves falseness. We use $=$ as the terms are not just definitionally equivalent, but syntactically identical.

Lemma 4.1 (False Preservation). $\mathbf{False}^\circ = \mathbf{False}$

To prove type preservation, we split the proof into three key lemmas. First, we show *compositionality*, *i.e.*, that the translation from CC-CC to CC commutes with substitution. Then we prove preservation of reduction semantics and equivalence, which essentially follows from compositionality. Finally, we prove type preservation, which relies on preservation of equivalence and on compositionality. The proofs are straightforward, since the typing rules in CC-CC essentially correspond to partial application already, so we elide them here. They follow the same structure as our type preservation proof for closure conversion, which we present in Section 5. For complete details, see our online technical appendix [14].

Compositionality is an important lemma since the type system and conversion relations are defined by substitution.

Lemma 4.2 (Compositionality). $(e[e'/x])^\circ = e^\circ[e'^\circ/x]$

Next we show that the translation preserves reduction, or that our model in CC weakly simulates reduction in CC-CC. This is used both to show that equivalence is preserved, since equivalence is defined by reduction, and to show type safety.

Lemma 4.3 (Pres. of Reduction). *If* $e \triangleright e'$ *then* $e^\circ \triangleright^* e'^\circ$

Now we show that reduction *sequences* are preserved. This essentially follows from preservation of single-step reduction, Lemma 4.3.

Lemma 4.4 (Preservation of Reduction Sequences). *If* $e \triangleright^* e'$ *then* $e^\circ \triangleright^* e'^\circ$

Next, we show *coherence*, *i.e.*, that the translation preserves equivalence. The proof essentially follows from Lemma 4.4, but we must show that our η rule for closures is preserved.

Lemma 4.5 (Coherence). *If* $e_1 \equiv e_2$ *then* $e_1^\circ \equiv e_2^\circ$

We can now show our final lemma: type preservation.

Lemma 4.6 (Type Preservation).

1. *If* $\vdash \Gamma$ *then* $\vdash \Gamma^\circ$
2. *If* $\Gamma \vdash e : A$ *then* $\Gamma^\circ \vdash e^\circ : A^\circ$

Finally, we can prove the desired consistency and type safety theorems.

Theorem 4.7 (Consistency of CC-CC). *There does not exist a closed expression* e *such that* $\cdot \vdash e : \mathbf{False}$.

Type safety tells us that there is no undefined behavior that causes a program to get stuck before it produces a value, and all programs terminate.

Theorem 4.8 (Type Safety of CC-CC). *If* $\cdot \vdash e : A$, *then* $e \triangleright^* v$ *and* $v \not\triangleright v'$.

5 Closure Conversion

We present the closure conversion translation in Figure 9. We define the following notation for the translation of expressions.

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

$$\frac{}{\Gamma \vdash \star : \Box \rightsquigarrow \star}\ [\text{CC-}*] \qquad \frac{}{\Gamma \vdash x : A \rightsquigarrow \mathbf{x}}\ [\text{CC-Var}] \qquad \frac{\Gamma \vdash e : A \rightsquigarrow \mathbf{e} \quad \Gamma \vdash A : U \rightsquigarrow \mathbf{A} \quad \Gamma, x : A \vdash e' : B \rightsquigarrow \mathbf{e'}}{\Gamma \vdash \text{let } x = e : A \text{ in } e' : B[e/x] \rightsquigarrow \mathbf{let\ x = e : A\ in\ e'}}\ [\text{CC-Let}]$$

$$\frac{\Gamma \vdash A : U \rightsquigarrow \mathbf{A} \quad \Gamma, x : A \vdash B : \star \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \star \rightsquigarrow \mathbf{\Pi\, x : A.\, B}}\ [\text{CC-Prod-}*] \qquad \frac{\Gamma \vdash A : U \rightsquigarrow \mathbf{A} \quad \Gamma, x : A \vdash B : \Box \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \Box \rightsquigarrow \mathbf{\Pi\, x : A.\, B}}\ [\text{CC-Prod-}\Box]$$

$$\frac{\Gamma \vdash A : U \rightsquigarrow \mathbf{A} \quad \Gamma, x : A \vdash B : U \rightsquigarrow \mathbf{B} \quad \begin{array}{c}\Gamma, x : A \vdash e : B \rightsquigarrow \mathbf{e} \\ x_i : A_i \ldots = \text{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma) \end{array} \quad \Gamma \vdash A_i : U \rightsquigarrow \mathbf{A}_i \ldots}{\begin{array}{l}\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B \rightsquigarrow \langle\!\langle (\mathbf{\lambda\, (n : \Sigma\, (x_i : A_i \ldots), x : let\ \langle x_i \ldots\rangle = n\ in\ A).} \\ \quad \mathbf{let\ \langle x_i \ldots\rangle = n\ in\ e),} \\ \quad \mathbf{\langle x_i \ldots\rangle\ as\ \Sigma\, (x_i : A_i \ldots)} \rangle\!\rangle \end{array}\ [\text{CC-Lam}]$$

$$\frac{\Gamma \vdash e_1 : \Pi\, x : A.\, B \rightsquigarrow \mathbf{e}_1 \quad \Gamma \vdash e_2 : A \rightsquigarrow \mathbf{e}_2}{\Gamma \vdash e_1\ e_2 : B[e_2/x] \rightsquigarrow \mathbf{e}_1\ \mathbf{e}_2}\ [\text{CC-App}] \qquad \frac{\Gamma \vdash A : \star \rightsquigarrow \mathbf{A} \quad \Gamma, x : A \vdash B : \star \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : \star \rightsquigarrow \mathbf{\Sigma\, x : A.\, B}}\ [\text{CC-Sig-}*]$$

$$\frac{\Gamma \vdash A : \Box \rightsquigarrow \mathbf{A} \quad \Gamma, x : A \vdash B : \Box \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : \star \rightsquigarrow \mathbf{\Sigma\, x : A.\, B}}\ [\text{CC-Sig-}\Box] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow \mathbf{e}}{\Gamma \vdash \text{fst}\, e : A \rightsquigarrow \mathbf{fst\ e}}\ [\text{CC-Fst}] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow \mathbf{e}}{\Gamma \vdash \text{snd}\, e : B[\text{fst}\, e/x] \rightsquigarrow \mathbf{snd\ e}}\ [\text{CC-Snd}]$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow \mathbf{e}}{\Gamma \vdash e : B \rightsquigarrow \mathbf{e}}\ [\text{CC-Conv}]$$

$\boxed{\vdash \Gamma \rightsquigarrow \mathbf{\Gamma}}$ where $\vdash \Gamma$

$$\frac{}{\vdash \cdot \rightsquigarrow \cdot}\ [\text{W-Empty}] \qquad \frac{\vdash \Gamma \rightsquigarrow \mathbf{\Gamma} \quad \Gamma \vdash A : \_ \rightsquigarrow \mathbf{A}}{\vdash \Gamma, x : A \rightsquigarrow \mathbf{\Gamma, x : A}}\ [\text{W-Assum}] \qquad \frac{\vdash \Gamma \rightsquigarrow \mathbf{\Gamma} \quad \Gamma \vdash A : \_ \rightsquigarrow \mathbf{A} \quad \Gamma \vdash e : A \rightsquigarrow \mathbf{e}}{\vdash \Gamma, x = e : A \rightsquigarrow \mathbf{\Gamma, x = e : A}}\ [\text{W-Def}]$$

Figure 9. Closure Conversion

$$e^+ \stackrel{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash e : A \rightsquigarrow \mathbf{e}$$

The CC-CC expression e^+ refers to the translation of the well-typed CC term e, with typing derivation for e as an implicit parameter.

Every case of the translation except for functions is trivial, including application [CC-App], since application is still the elimination form for closures after closure conversion. In the nontrivial case [CC-Lam], we translate CC functions to CC-CC closures, as described in Section 3. The translation of a function $\lambda\, x : A.\, e$ produces a closure $\langle\!\langle \mathbf{e}_1, \mathbf{e}_2 \rangle\!\rangle$. We compute the free variables (and their type annotations) of the function $\lambda\, x : A.\, e$, $x_i : A_i \ldots$, using the metafunction $\text{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma)$ defined shortly. The first component $\mathbf{e}_1$ is closed code. Ignoring the type annotation for a moment, the code $\mathbf{\lambda\, (n, x).\ let\ \langle x_i \ldots\rangle = n\ in}\ e^+$ projects each of the $|i|$ free variables $\mathbf{x}_i \ldots$ from the environment $\mathbf{n}$ and binds them in the scope of the body e^+. But CC-CC is dependently typed, so we also bind the free variables from the environment in the type annotation for the argument $\mathbf{x}$, *i.e.*, producing the annotation $\mathbf{x : let\ \langle x_i \ldots\rangle = n\ in}\ A^+$ instead of just $\mathbf{x} : A^+$. Next we produce the environment type $\Sigma\, (x_i : A^+ \ldots)$, from the free source variables $x_i \ldots$ of types $A_i \ldots$. We create the environment $\mathbf{e}_2$ by creating the dependent n-tuple $\langle \mathbf{x}_i \ldots \rangle$; these free variables will be replaced by values at run time.

$$\begin{array}{lcll} \text{FV}(e, B, \Gamma) & \stackrel{\text{def}}{=} & \Gamma_0, \ldots, \Gamma_n, x_0 : A_0, \ldots, x_n : A_n \\ & \textit{where} & x_0, \ldots, x_n = \text{fv}(e, B) \\ & & \Gamma \vdash x_0 : A_0, \ldots, \Gamma \vdash x_n : A_n \\ & & \Gamma_0 = \text{FV}(A_0, \_, \Gamma) \\ & & \quad \vdots \\ & & \Gamma_n = \text{FV}(A_n, \_, \Gamma) \end{array}$$

Figure 10. CC Dependent Free Variable Sequences

To compute the sequence of free variables and their types, we define the metafunction $\text{FV}(e, B, \Gamma)$ in Figure 10. Just from the syntax of terms e, B, we can compute some sequence of free variables $x_0, \ldots, x_n = \text{fv}(e, B)$. However, the types of these free variables $A_0, \ldots, A_n$ may contain *other* free variables, and their types may contain still others, and so on! We must, therefore, recursively compute the a sequence of free variables and their types with respect to an environment Γ. Note that because the type B of a term e may contain different free variables than the term, we must compute the sequence with respect to both a term and its type. However, in all recursive applications of this metafunction—*e.g.*, $\text{FV}(A_0, \_, \Gamma)$—the type of A_0 must be a universe and cannot have any free variables.

5.1 Type Preservation

First we prove type preservation, using the same staging as in Section 4. After we show type preservation, we show

correctness of separate compilation. In CC, the lemmas required for type preservation do most of the work to allow us to prove correctness of separate compilation, since type checking includes reduction and thus we prove preservation of reduction sequences.

We first show *compositionality*. This lemma, which establishes that translation commutes with substitution, is the key difficulty in our proof of type preservation because closure conversion internalizes free variables. Whether we substitute a term for a variable before or after translation can drastically affect the shape of closures produced by the translation. For instance, consider the term $(\lambda\, y : A.\, e)[e'/x]$. If we perform this substitution before translation, then we will generate an environment with the shape $\langle x_i \ldots, x_j \ldots\rangle$, *i.e.*, with only free variables and without x in the environment. However, if we translate the individual components and then perform the substitution, then the environment will have the shape $\langle x_i \ldots, e'^{+}, x_j \ldots\rangle$—that is, x would be free when we create the environment and substitution would replace it by e'^{+}. We use our η-principle for closures to show that closures that differ in this way are still equivalent.

Lemma 5.1 (Compositionality). $(e_1[e_2/x])^+ \equiv e_1^+[e_2^+/x]$

Proof. By induction on the typing derivation for e_1. We give the key cases.

Case [Ax-Var]
We know that e_1 is some free variable x', so either $x' = x$, hence $e_2^+ \equiv e_2^+$, or $x' \neq x$, hence $x'^{+} \equiv x'^{+}$.

Case [T-Code-*]
We know that $e_1 = \Pi\, x' : A.\, B$. W.l.o.g., assume $x' \neq x$. We must show $(\Pi\, x' : A[e_2/x].\, B[e_2/x])^+ \equiv (\Pi\, x' : A.\, B)^+[e_2^+/x]$.

$$(\Pi\, x' : A[e_2/x].\, B[e_2/x])^+ \tag{1}$$
$$= \Pi\, x' : (A[e_2/x])^+.\, (B[e_2/x])^+ \tag{2}$$
by definition of the translation
$$= \Pi\, x' : (A^+[e_2^+/x]).\, (B^+[e_2^+/x]) \tag{3}$$
by the inductive hypothesis for A and B
$$= (\Pi\, x' : A^+.\, B^+)[e_2^+/x] \tag{4}$$
by definition of substitution
$$= (\Pi\, x' : A.\, B)^+[e_2^+/x] \tag{5}$$
by definition of translation

Case [Lam]
We know that $e_1 = \lambda\, y : A.\, e$. W.l.o.g., assume that $y \neq x$. We must show that $((\lambda\, y : A.\, e)[e_2/x])^+ \equiv (\lambda\, y : A.\, e)^+[e_2^+/x]$. Recall that by convention we have that $\Gamma \vdash \lambda\, y : A.\, e : \Pi\, y : A.\, B$.

$$((\lambda\, y : A.\, e)[e_2/x])^+ \tag{6}$$
$$= (\lambda\, y : (A[e_2/x]).\, e[e_2/x])^+ \tag{7}$$
by substitution
$$= \langle\!\langle(\mathbf{\lambda}\, n : \Sigma\, (x_i : A_i^+ \ldots),\, y : \mathbf{let}\, \langle x_i \ldots\rangle = n\, \mathbf{in}\, (A[e_2/x])^+.\ \mathbf{let}\, \langle x_i \ldots\rangle = n\, \mathbf{in}\, (e[e_2/x])^+),\, \langle x_i \ldots\rangle\rangle\!\rangle \tag{8}$$
by definition of the translation
where $x_i : A_i \ldots = \mathrm{FV}(\lambda\, y : (A[e_2/x]).\, e[e_2/x], \Gamma)$. Note that x is not in the sequence $(x_i \ldots)$.
On the other hand, we have

$$f = (\lambda\, y : A.\, e)^+[e_2^+/x] \tag{9}$$
$$= \langle\!\langle(\mathbf{\lambda}\, n : \Sigma\, (x_j : A_j^+ \ldots),\, y : \mathbf{let}\, \langle x_j \ldots\rangle = n\, \mathbf{in}\, A^+.\ \mathbf{let}\, \langle x_j \ldots\rangle = n\, \mathbf{in}\, e^+),\, \langle x_{j_0} \ldots, e_2^+, x_{j_{i+1}} \ldots\rangle\rangle\!\rangle \tag{10}$$
by definition of the translation
where $x_j : A_j \ldots = \mathrm{FV}(\lambda\, y : A.\, e, \Gamma)$. Note that x is in $x_j \ldots$; we can write the sequence as $(x_{j_0} \ldots x, x_{j_{i+1}} \ldots)$. Therefore, the environment we generate contains e_2^+ in position j_i.
By $[\equiv\text{-Clo}_1]$, it suffices to show that $\mathbf{let}\, \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\, \mathbf{in}\, (e[e_2/x])^+ \equiv f\ y$ where f is the closure from Equation (9).

$$f\ y \equiv \mathbf{let}\, \langle x_{j_0} \ldots x, x_{j_{i+1}} \ldots\rangle = \langle x_{j_0} \ldots, e_2^+, x_{j_{i+1}} \ldots\rangle\, \mathbf{in}\, e^+ \tag{11}$$
by $\triangleright_\beta$ in CC-CC
$$\equiv e^+[e_2^+/x] \tag{12}$$
by $|j|$ applications of $\triangleright_\zeta$, since only x has a value
$$\equiv (e[e_2/x])^+ \tag{13}$$
by the inductive hypothesis applied to the derivation for e
$$\equiv \mathbf{let}\, \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\, \mathbf{in}\, (e[e_2/x])^+ \tag{14}$$
by $|i|$ applications of $\triangleright_\zeta$, since no variable has a value □

Next we show that if a source term e takes a step, then its translation e^+ reduces in some number of steps to a definitionally equivalent term e. This proof essentially follows by Lemma 5.1. Then we show by induction on the length of the reduction sequence that the translation preserves reduction sequences. Note that since Lemma 5.1 relies on our η equivalence rule for closures, we can only show reduction up to definitional equivalence. That is, we cannot show $e^+ \triangleright^* e'^{+}$. This is not a problem; we reason about source programs to equivalence anyway, and not up to syntactic equality.

Lemma 5.2 (Preservation of Reduction). *If* $\Gamma \vdash e \triangleright e'$ *then* $\Gamma^+ \vdash e^+ \triangleright^* e$ *and* $e \equiv e'^{+}$

Proof. By cases on $\Gamma \vdash e \triangleright e'$. Most cases follow easily by Lemma 5.1, since most cases of reduction are defined by substitution. We give representative cases; see our online technical appendix [14].

Case $(\lambda\, x : A.\, e_1)\, e_2 \triangleright_\beta e_1[e_2/x]$
We must show that $((\lambda\, x{:}A.\, e_1)\, e_2)^+ \triangleright^* e$ and $(e_2[e_1/x])^+ \equiv e$. Let $e \stackrel{\text{def}}{=} e_1^+[e_2^+/x]$.
By definition of the translation, $((\lambda\, x : A.\, e_1)\, e_2)^+ = f\ e_2^+$, where

$$f = \langle\!\langle(\mathbf{\lambda}\, n : \Sigma\, (x_i : A_i^+ \ldots),\, x : \mathbf{let}\, \langle x_i \ldots\rangle = n\, \mathbf{in}\, A^+. \tag{15}$$
$$\mathbf{let}\, \langle x_i \ldots\rangle = n\, \mathbf{in}\, e_1^+),\, \langle x_i \ldots\rangle\rangle\!\rangle \tag{16}$$
and where $x_i : A_i \ldots = \mathrm{FV}(\lambda\, x : A.\, e_1, \Gamma)$.

To complete the proof, observe that,

$$\mathbf{f}\ e_2^+ \rhd_\beta \mathbf{let}\ \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\ \mathbf{in}\ e_1^+[e_2^+/x] \quad (17)$$

$$\rhd_\zeta^{|i|}\ e_1^+[e_2^+/x] \quad (18)$$

$$\equiv (e_1[e_2/x])^+ \quad \text{by Lemma 5.1} \quad (19)$$

□

Lemma 5.3 (Preservation of Reduction Sequences). *If* $\Gamma \vdash e \rhd^* e'$ *then* $\Gamma^+ \vdash e^+ \rhd^* e$ *and* $\Gamma^+ \vdash e \equiv e'^+$.

We can now show *coherence*, *i.e.*, that equivalent terms are translated to equivalent terms. As equivalence is defined primarily by $\rhd^*$, the only interesting part of the next proof is preserving η equivalence. To show that η equivalence is preserved, we require our new η rules for closures.

Lemma 5.4 (Coherence). *If* $\Gamma \vdash e \equiv e'$, *then* $\Gamma^+ \vdash e^+ \equiv e'^+$.

Proof. By induction on the $e \equiv e'$ judgment.

Case [$\equiv$-η_1]

By assumption, $e \rhd^* \lambda\, x : t.\, e_1$, $e' \rhd^* e_2$ and $e_1 \equiv e_2\ x$.
Must show $e^+ \equiv e'^+$.
By Lemma 5.3, $e^+ \rhd^* \mathbf{e}$ and $\mathbf{e} \equiv (\lambda\, x{:}t.\, e_1)^+$, and similarly $e'^+ \rhd^* \mathbf{e'}$ and $\mathbf{e'} \equiv e_2^+$.
By transitivity of $\equiv$, it suffices to show $(\lambda\, x : t.\, e_1)^+ \equiv e_2^+$.
By definition of the translation,

$$(\lambda\, x : t.\, e_1)^+ = \langle\!\langle (\mathbf{\lambda}\, \mathbf{n} : \Sigma\, (x_i : A_i^+ \ldots), \mathbf{x} : \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ A^+.\ \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ e_1^+), \langle x_i \ldots\rangle \rangle\!\rangle$$

where $x_i : A_i \ldots = \mathrm{FV}(\lambda\, x : t.\, e_1, \Gamma)$.
By [$\equiv$-Clo$_1$] in CC-CC, it suffices to show that

$$\mathbf{let}\ \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\ \mathbf{in}\ e_1^+ \quad (20)$$

$$\equiv e_1^+ \quad (21)$$

by |i| applications of $\rhd_\zeta$

$$\equiv e_2^+\ \mathbf{x} \quad (22)$$

by the inductive hypothesis applied to $e_1 \equiv e_2\ x$ □

Now we can prove type preservation. We give the technical version of the lemma required to complete the proof, followed by the desired statement of the theorem.

Lemma 5.5 (Type Preservation (technical)).

1. *If* $\vdash \Gamma$ *then* $\vdash \Gamma^+$
2. *If* $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash e^+ : A^+$

Proof. Parts 1 and 2 proven simultaneously by induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash e : A$.

Part 1 follows easily by induction and part 2. We give the key cases for part 2.

Case [Lam]

We have that $\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B$. We must show that $\Gamma^+ \vdash (\lambda\, x : A.\, e)^+ : (\Pi\, x : A.\, B)^+$.
By definition of the translation, we must show that

$$\langle\!\langle (\mathbf{\lambda}\, (\mathbf{n} : \Sigma\, (x_i : A_i^+ \ldots), \mathbf{x} : \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ A^+).\ :\ \mathbf{\Pi}\, \mathbf{x} : A^+.\, B^+ \quad \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ e_1^+), \langle x_i \ldots\rangle \rangle\!\rangle$$

where $x_i : A_i \ldots = \mathrm{FV}(\lambda\, x : t.\, e_1, \Gamma)$.
Notice that the annotation in the term $\mathbf{x} : \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ A^+$, does not match the annotation in the type $\mathbf{x} : A^+$. However, by [Clo], we can derive that the closure has type:
$\mathbf{\Pi}\, (\mathbf{x}{:}\mathbf{let}\ \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\ \mathbf{in}\ A^+).\ (\mathbf{let}\ \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\ \mathbf{in}\ B^+)$,
This is equivalent to $\mathbf{\Pi}\, \mathbf{x} : A^+.\, B^+$ (under Γ^+), since $(\mathbf{let}\ \langle x_i \ldots\rangle = \langle x_i \ldots\rangle\ \mathbf{in}\ A^+) \equiv A^+$ as we saw in earlier proofs. So, by [Clo] and [Conv], it suffices to show that the environment and the code are well-typed.
By part 1 of the induction hypothesis applied (since each of $x_i : A_i \ldots$ come from Γ), we know the environment is well-typed: $\Gamma^+ \vdash \langle x_i \ldots\rangle : \Sigma\, (x_i : A_i^+ \ldots)$.
Now we show that the code

$$(\mathbf{\lambda}\, (\mathbf{n} : \Sigma\, (x_i : A_i^+ \ldots), \mathbf{x} : \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ A^+).\ \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ e_1^+)$$

has type $\mathbf{Code}\, (\mathbf{n}, \mathbf{x}).\ \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ B^+$. For brevity, we omit the duplicate type annotations on $\mathbf{n}$ and $\mathbf{x}$.
Observe that by the induction hypothesis applied to $\Gamma \vdash A : U$ and by weakening

$$\mathbf{n} : \Sigma\, (x_i : A_i^+ \ldots) \vdash \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ A^+ : U^+.$$

Hence, by [Code], it suffices to show

$$\cdot, \mathbf{n}, \mathbf{x} \vdash \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ e_1^+ : \mathbf{let}\ \langle x_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ B^+$$

which follows by the inductive hypothesis applied to $\Gamma, x : A \vdash e_1 : B$, and by weakening, since $x_i \ldots$ are the free variables of e_1, A, and B.

Case [App]

We have that $\Gamma \vdash e_1\ e_2 : B[e_2/x]$. We must show that $\Gamma^+ \vdash e_1^+\ e_2^+ : (B[e_2/x])^+$. By Lemma 5.1, it suffices to show $\Gamma^+ \vdash e_1^+\ e_2^+ : B^+[e_2^+/x]$, which follows by [App] and the inductive hypothesis applied to e_1, e_2 and B. □

Theorem 5.6 (Type Preservation). *If* $\Gamma \vdash e : t$ *then* $\Gamma^+ \vdash e^+ : t^+$.

5.2 Correctness

We prove *correctness of separate compilation* and *whole program correctness*. These two theorems follow easily from Lemma 5.3, but requires a little more work to state formally.

First, we need an independent specification that relates source values to target values in CC-CC. We do this by adding ground types, such as Bool, to both languages and consider results related when they are the same boolean: $\mathbf{true} \approx \mathbf{true}$ and $\mathbf{false} \approx \mathbf{false}$. It is well known how specify more sophisticated notions of observations, and we discuss these in Section 6.

Next, we define components and linking. Components in both CC and CC-CC are well-typed open terms, *i.e.*, $\Gamma \vdash e : A$. We implement linking by substitution, and define valid closing substitutions γ as follows.

$$\Gamma \vdash \gamma \quad \stackrel{\text{def}}{=} \quad \forall x : A \in \Gamma, \cdot \vdash \gamma(x) : A$$

We extend the compiler to closing substitutions γ^+ by pointwise application of the translation.

Our separate compilation guarantee is that the translation of the source component e linked with substitution γ is equivalent to first compiling e and then linking with some $\boldsymbol{\gamma}$ that is definitionally equivalent to γ^+.

Theorem 5.7 (Correctness of Separate Compilation). *If* $\Gamma \vdash e : A$ *and* A *is a ground type,* $\Gamma \vdash \gamma$, $\Gamma^+ \vdash \boldsymbol{\gamma}$, $\gamma(e) \rhd^* v$, *and* $\gamma^+ \equiv \boldsymbol{\gamma}$ *then* $\boldsymbol{\gamma}(e^+) \rhd^* v'$ *and* $v^+ \approx v'$

Proof. Since the translation commutes with substitution, preserves equivalence, reduction implies equivalence, and equivalence is transitive, the following diagram commutes.

$$\begin{array}{ccc} (\gamma(e))^+ & \xrightarrow{\equiv} & \boldsymbol{\gamma}(e^+) \\ \downarrow{\scriptstyle\equiv} & & \downarrow{\scriptstyle\equiv} \\ v^+ & \xrightarrow{\equiv} & v' \end{array}$$

Since $\equiv$ on ground types implies $\approx$, we know that $v \approx v'$. □

As a simple corollary, our compiler must also be whole-program correct. If a whole-program e evaluates to a value v, then the translation e^+ runs to a value equivalent to v^+.

Corollary 5.8 (Whole-Program Correctness). *If* $\cdot \vdash e : A$ *and* A *is a ground type, and* $e \rhd^* v$ *then* $e^+ \rhd^* \mathbf{v}$ *and* $v^+ \approx \mathbf{v}$

6 Related Work and Discussion

Preserving Dependent Types Barthe et al. [9] study the call-by-name (CBN) CPS translation for the Calculus of Constructions without Σ types. In 2002, when attempting to extend the translation to CIC, Barthe and Uustalu [10] noticed that in the presence of Σ types, the standard typed CPS translation fails. In recent work, Bowman et al. [15] show how to recover type preservation for both CBN and call-by-value CPS. Bowman et al. [15] add a new typing rule that keeps track of additional contextual information while type checking continuations, similar to our [Clo] typing rule that keeps track of the environment (via substitution) while type checking closures. In CPS, a term $\mathbf{e}$ is evaluated to values indirectly, by a continuation $\lambda\, \mathbf{x}.\, \mathbf{e}'$. When resolving type equivalence, they end up requiring that $\mathbf{e}$ is equivalent to $\mathbf{x}$. An essentially similar problem arises in the translation Π types described in Section 3. In closure conversion, we need show that a free variable $\mathbf{x}$ is the same as a projection from an environment $\mathbf{fst\ n}$ when resolving type equivalence.

There is also work on typed compilation of restricted forms of dependency, which avoid the central type theoretic difficulties we solve. Chen et al. [16] develop a type-preserving compiler from Fine, an ML-like language with refinement types, to a version of the .NET intermediate language with type-level computation. This system lacks full spectrum types and can rely on computational irrelevance of type-level arguments, unlike our setting as discussed in Section 3. Shao et al. [42] use CIC as an *extrinsic type system* over an ML-like language, so that CIC terms can be used as specifications and proofs, but restrict arbitrary terms from appearing in types. They develop a type-preserving closure conversion translation for this language. Their closure conversion is simpler to develop than ours, because the extrinsic type system avoids the issues of computational relevance by disallowing terms from appear in types. Instead, a separate type-level representation of a subset of terms can appear in types. This can increase the burden of proving programs correct compared to an intrinsic system such as CC because the programmer is required to duplicate programs into the type-level representation.

Separate and Compositional Compilation In this work, we prove Theorem 5.7 (Correctness of Separate Compilation) This is similar to the guarantees of SepCompCert [24]. We could support a *compositional* compiler correctness result by developing a relation independent of the compiler between source and target components to classify which components are safe to link with. There are well known techniques for developing such relations which we think will extend to CC [13, 36, 37, 41, 43].

Type-Preserving Compilation Type-preserving compilation has been widely used to rule out linking errors, and even extended to statically rule out security attacks introduced by linking. The seminal work by Morrisett et al. [35] uses type-preserving compilation and give a safe linking semantics to Typed Assembly Languages (TAL): linking any two TAL components—regardless of whether they were generated by a correct compiler or hand-written—is guaranteed to be type and memory safe. Our target language CC-CC provides similar guarantees to TAL, although it is still a high-level language by comparison.

7 Future Work

The Calculus of Inductive Constructions As future work, we plan to scale our translation to the Calculus of Inductive Constructions (CIC), the core language of Coq. There are two key challenges in scaling to CIC.

First, we need to scale our work to recursive functions. Our translation should scale easily, but adding recursion to CC-CC will be challenging. In CIC, recursive functions must always terminate to ensure consistency. Coq enforces this with a syntactic guard condition that cannot be preserved by closure conversion, as it relies on the structure of free variables. Instead, we intend to investigate two alternatives to ensuring termination in CC-CC. One is to compile the guard condition to inductive eliminators—essentially a primitive form that folds over the tree structure of an inductive type and is terminating by construction, which has been studied in Coq [18], but it is not clear how to encode this in a typed assembly. A more theoretically appealing technique is to design CC-CC with semantic termination concepts such as sized types [19]. However, it is not clear how to compile Coq programs based on the guard condition to sized types.

Recursion also introduces an important performance consideration. Abstract closure conversion introduces additional allocations and dereferences compared to the existential type translation [30, 34] To solve this, we need to adapt our definition of closures to enable environments to be separated from the closures, but still hide their type.

The second challenge is how to address computational relevance. In CC, it is simple to make a syntactic distinction between relevant and irrelevant terms [9] based on the universes $\star$ and $\Box$, although we avoid doing so for simplicity of the presentation. Coq features an infinite hierarchy of universes, so the distinction is not easy to make. We would need to design a source language in which computational relevance has already been made explicit so that we can easily decide which terms to closure-convert. Some work has been done on the design of such languages [7, 31–33], but to our knowledge none of the work has been used to encode Coq's computational relevance semantics.

Full-Spectrum Dependently Typed Assembly Language Our ultimate goal is to compile to a dependently typed assembly-like language in which we can safely link and then generate machine code. We imagine the assembly could be targeted from many languages, such as Coq and OCaml, and type checking in this assembly would serve to ensure safe linking. This would require a general purpose typed assembly, with support for interoperating between pure code and effectful code [39] and different type systems [1].

A minimal compiler for a functional language performs CPS translation, closure conversion, heap allocation, and assembly-code generation. Recent work solves CPS [15] and we solve closure conversion, so two passes remain.

Heap allocation makes memory and allocation explicit, so we need new typing and equivalence rules that can reason about memory and allocation. This seems straightforward for CC, but we anticipate further challenges for CIC. In CIC, we must allow cycles in the heap to support recursive functions but still ensure soundness and termination. As discussed earlier, ensuring terminating recursion alone will introduce new challenges, but other techniques may help us solve the problem once the heap is explicit. Linear types have been used to allow cycles in the heap but still guarantee strong normalization [4]. Unfortunately, linear types and dependent types are difficult to integrate [29].

The design of a dependently typed assembly language will be hard. Assembly language will make machine-level concepts explicit, such as registers, word sizes, and first-class control. We will need typing and equivalence rules to reason about these machine-level concepts. First-class control presents a particular challenge, since it is been shown inconsistent with dependent type theory [22]. This is related to the problem of CPS and dependent types, so we anticipate that we can build on the work of Bowman et al. [15] to restrict control and regain consistency. Even if control is not a problem, the consistency proof will introduce new challenges. In this work, we develop a model of the CC-CC in CC to prove type safety and consistency. This relies critically on *compositionality*. Assembly languages are typically not compositional, so this proof architecture may not scale to the assembly language. However, in recent work on interoperability between a high-level functional language and a typed assembly language, Patterson et al. [40] successfully defined a compositional assembly language, and we are hopeful that we can extend this work to the dependently typed setting.

Secure Compilation Type preservation has been widely studied to statically enforce *secure compilation*, *i.e.*, to statically guarantee that compiled code cannot be linked with components (attackers) that violate data hiding, abstraction, and information flow security properties [2, 3, 13, 37]. These compilers typically prove that the translation preserves and reflects *contextual equivalence*, *i.e.*, that the compiler is fully abstract. As future work, we plan to investigate what security guarantees can be implied just from preservation and reflection of *definitional equivalence*, which we conjecture holds of our translation.

In our model, we prove Lemma 4.5 (Coherence), *i.e.*, that we can translate any two *definitionally equivalent* CC-CC terms into definitionally equivalent CC terms. In our compiler, we prove Lemma 5.4 (Coherence), *i.e.*, that we translate any two definitionally equivalent CC terms into definitionally equivalent CC-CC terms. These two lemmas resemble the statements of preservation and reflection, although in terms of definitional equivalence instead of contextual equivalence. Since Lemma 4.5 is not stated in terms of our compiler, we need the following condition to complete the proof of preservation and reflection: $e \equiv (e^+)^\circ$, *i.e.*, that compiling to CC-CC and then translating back to CC is equivalent to the original term; we conjecture this equivalence holds.

Definitional equivalence in dependently typed languages is sound, but not necessarily complete, with respect to contextual equivalence, so even if the above conjecture holds, more work remains to prove full abstraction. Typed closure conversion based on the existential-type encoding is well known to be fully abstract [2, 37]. We conjecture that our translation is also fully abstract; the essential observation is that CC-CC does not include any constructs that would allow an attacker to inspect the environment.

Acknowledgments

We gratefully acknowledge the valuable feedback provided by Greg Morrisett, Stephanie Weirich, and the anonymous reviewers. We also give special thanks to Dan Grossman, without whom this paper would not have been. Part of this work was done at Inria Paris in Fall 2017, while Amal Ahmed was a Visiting Professor and William J. Bowman held an internship. This material is based upon work supported by the National Science Foundation under grants CCF-1422133 and CCF-1453796, and the European Research Council under ERC Starting Grant SECOMP (715753). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our funding agencies.

References

[1] Amal Ahmed. 2015. Verified Compilers for a Multi-language World. In *Summit on Advances in Programming Languages (SNAPL)*, Vol. 32. https://doi.org/10.4230/LIPIcs.SNAPL.2015.15

[2] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1411204.1411227

[3] Amal Ahmed and Matthias Blume. 2011. An Equivalence-preserving CPS Translation Via Multi-language Semantics. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2034773.2034830

[4] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3: A Linear Language with Locations. *Fundamenta Informaticae* 77, 4 (Dec. 2007). https://doi.org/10.1007/11417170_22

[5] Abhishek Anand, A. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A Verified Compiler for Coq. In *International Workshop on Coq for Programming Languages (CoqPL)*. http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf

[6] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 2 (April 2015). https://doi.org/10.1145/2701415

[7] Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions As a Programming Language with Dependent Types. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. https://doi.org/10.1007/978-3-540-78499-9_26

[8] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1480881.1480894

[9] 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 (Sept. 1999). https://doi.org/10.1023/a:1010000206149

[10] Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. In *Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*. https://doi.org/10.1145/509799.503043

[11] Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for Free: Parametricity for Dependent Types. *Journal of Functional Programming (JFP)* 22, 02 (March 2012). https://doi.org/10.1017/S0956796812000056

[12] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Conference on Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3018610.3018620

[13] William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2784731.2784733

[14] William J. Bowman and Amal Ahmed. 2018. *Typed Closure Conversion for the Calculus of Constructions (Technical Appendix)*. Technical Report. https://www.williamjbowman.com/resources/cccc-techrpt.pdf

[15] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-preserving CPS Translation of Σ and Π Types Is Not Not Possible. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL (Jan. 2018). https://doi.org/10.1145/3158110

[16] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving Compilation of End-to-end Verification of Security Enforcement. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1806596.1806643

[17] Thierry Coquand. 1986. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science (LICS)*. https://hal.inria.fr/inria-00076023

[18] Eduardo Giménez. 1995. Codifying Guarded Definitions with Recursive Schemes. In *International Workshop on Types for Proofs and Programs (TYPES)*. https://doi.org/10.1007/3-540-60579-7_3

[19] Benjamin Grégoire and Jorge Luis Sacchini. 2010. On Strong Normalization of the Calculus of Constructions with Type-based Termination. In *International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. https://doi.org/10.1007/978-3-642-16242-8_24

[20] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (newman) Wu, Shu-chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2775051.2676975

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[22] Hugo Herbelin. 2005. On the Degeneracy of Σ-types in Presence of Computational Classical Logic. In *International Conference on Typed Lambda Calculi and Applications*. https://doi.org/10.1007/11417170_16

[23] James G. Hook and Douglas J. Howe. 1986. *Impredicative Strong Existential Equivalent to Type:Type*. Technical Report. Cornell University. http://hdl.handle.net/1813/6600

[24] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837642

[25] Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *International Workshop on Computer Science Logic (CSL)*. https://hal.inria.fr/hal-00730913

[26] Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *International Workshop on Computer Science Logic (CSL)*. https://doi.org/10.4230/LIPIcs.CSL.2013.432

[27] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (Nov. 2009). https://doi.org/10.1007/s10817-009-9155-4

[28] Zhaohui Luo. 1989. ECC, an Extended Calculus of Constructions. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/lics.1989.39193

[29] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. https://doi.org/10.1007/978-3-319-30936-1_12

[30] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/237721.237791

[31] Alexandre Miquel. 2001. The Implicit Calculus of Constructions. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*. https://doi.org/10.1007/3-540-45413-6_27

[32] Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. https://doi.org/10.1007/978-3-540-78499-9_25

[33] Richard Nathan Mishra-linger. 2008. *Irrelevance, Polymorphism, and Erasure in Type Theory*. Ph.D. Dissertation. Portland State University. http://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=3678&context=open_access_etds

[34] Greg Morrisett and Robert Harper. 1998. Typed Closure Conversion for Recursively-defined Functions. *Electronic Notes in Theoretical Computer Science* 10 (June 1998), 230–241. https://doi.org/10.1016/s1571-0661(05)80702-9

[35] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/268946.268954

[36] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2784731.2784764

[37] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation Via Universal Embedding. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2951913.2951941

[38] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP (Aug. 2017). https://doi.org/10.1145/3110276

[39] Daniel Patterson and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *Summit on Advances in Programming Languages (SNAPL)*. https://doi.org/10.4230/LIPIcs.SNAPL.2017.12

[40] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062347

[41] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-54833-8_8

[42] Zhong Shao, Valery Trifonov, Bratin Saha, and Nikolaos Papaspyrou. 2005. A Type System for Certified Binaries. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 1 (Jan. 2005). https://doi.org/10.1145/1053468.1053469

[43] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2676985

[44] The Coq Development Team. 2017. The Coq Proof Assistant Reference Manual. https://web.archive.org/web/20170109225110/https://coq.inria.fr/doc/Reference-Manual006.html