Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible

William J. Bowman, Youyou Cong, Nick Rioux, Amal Ahmed

Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible*

WILLIAM J. BOWMAN, Northeastern University, USA
YOUYOU CONG, Ochanomizu University, Japan
NICK RIOUX, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

Dependently typed languages such as Coq are used to specify and prove functional correctness of source programs, but what we ultimately need are guarantees about correctness of compiled code. By preserving dependent types through each compiler pass, we could preserve source-level specifications and correctness proofs into the generated target-language programs. Unfortunately, type-preserving compilation of dependent types is hard. In 2002, Barthe and Uustalu showed that type-preserving CPS is *not possible* for languages such as Coq. Specifically, they showed that for strong dependent pairs (Σ types), the standard typed call-by-name CPS is *not type preserving*. They further proved that for dependent case analysis on sums, a class of typed CPS translations—including the standard translation—is *not possible*. In 2016, Morrisett noticed a similar problem with the standard call-by-value CPS translation for dependent functions (Π types). In essence, the problem is that the standard typed CPS translation by double-negation, in which computations are assigned types of the form $(A \to \bot) \to \bot$, disrupts the term/type equivalence that is used during type checking in a dependently typed language.

In this paper, we prove that type-preserving CPS translation for dependently typed languages is *not* not possible. We develop both call-by-name and call-by-value CPS translations from the Calculus of Constructions with both Π and Σ types (CC) to a dependently typed target language, and prove type preservation and compiler correctness of each translation. Our target language is CC extended with an additional equivalence rule and an additional typing rule, which we prove consistent by giving a model in the extensional Calculus of Constructions. Our key observation is that we can use a CPS translation that employs *answer-type polymorphism*, where CPS-translated computations have type $\forall\alpha.(A \to \alpha) \to \alpha$. This type justifies, by a *free theorem*, the new equality rule in our target language and allows us to recover the term/type equivalences that CPS translation disrupts. Finally, we conjecture that our translation extends to dependent case analysis on sums, despite the impossibility result, and provide a proof sketch.

CCS Concepts: • **Software and its engineering → Correctness**; **Functional languages**; **Polymorphism**; **Control structures**; **Compilers**; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: Dependent types, type theory, secure compilation, type-preserving compilation, parametricity, CPS

22

*We use a non-bold blue sans-serif font to typeset the source language, and a **bold red serif font** for the target language. The languages are distinguishable in black-and-white, but the paper is easier to read when viewed or printed in color.

Authors' addresses: William J. Bowman, Northeastern University, USA, wjb@williamjbowman.com; Youyou Cong, Ochanomizu University, Japan, so.yuyu@is.ocha.ac.jp; Nick Rioux, Northeastern University, USA, rioux.n@husky.neu.edu; Amal Ahmed, Northeastern University, USA, amal@ccs.neu.edu.

2475-1421/2018/1-ART22
https://doi.org/10.1145/3158110

ACM Reference Format:
William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article 22 (January 2018), 33 pages. https://doi.org/10.1145/3158110

1 INTRODUCTION

Dependently typed languages such as Coq's Gallina have had tremendous impact on the state of the art in fully verified software in recent years. Such languages enable verification of program properties alongside program development, a strategy that has been used to verify full functional correctness of a range of software, including the CompCert C compiler [Leroy 2006, 2009], the CertiKOS OS kernel [Gu et al. 2015], and implementations of cryptographic protocols [Appel 2015; Barthe et al. 2009]. But, while dependently typed languages make it easier to verify properties of *source* (Gallina) programs, what is ultimately needed is a guarantee that the same properties hold of compiled *low-level* code. This calls for a verified compiler for Gallina and the work underway on CertiCoq [Anand et al. 2017] is a good first step. However, CertiCoq erases Gallina types and then performs transformations such as CPS and closure conversion on untyped code. The problem with erasing types too early in the compilation pipeline[1] is that it becomes difficult to build a verified compiler that supports safe/secure *linking* of compiled code with code from other languages [Ahmed 2015; Patterson and Ahmed 2017]. For instance, consider a (pure) higher-order Gallina function `f` that is compiled to C or assembly (as in CertiCoq) and linked with a target context that passes the compiled `f` an impure C or assembly "function" as input. This impure function may break Gallina's type-abstraction or security guarantees or modify `f`'s control flow.

Investigating *type-preserving compilation of dependently typed languages* is critical because, intuitively, the key to protecting the compiled version of `f` from contexts that provide ill-behaved inputs is to ensure that compiled code can only be linked with target-level contexts that *correspond* to well-typed source-level contexts. By translating types during compilation, we can encode that correspondence in the types. Then, at link time, we allow linking a well-typed compiled component with other well-typed target language components. With type-preserving compilation and sufficiently rich types at the target level, we can statically rule out linking with ill-behaved/insecure contexts. Thus, type-preserving compilation provides a path to building secure (fully abstract) compilers [Abadi 1998; Ahmed and Blume 2008, 2011; Bowman and Ahmed 2015; Fournet et al. 2013; Kennedy 2006; New et al. 2016; Patrignani et al. 2015] without the overhead of dynamic checks. By preserving dependent types, we can even preserve the full functional specifications into the target level, so that compiled code can be independently verified by type checking.

This paper investigates type-preserving CPS for dependently typed languages. CPS translation is an important compiler pass that makes control flow and evaluation order explicit[2]. However, it is a transformation that presents nontrivial problems in a dependently typed setting, as discussed next.

Prior Work. Barthe et al. [1999] showed how to scale typed call-by-name (CBN) CPS translation to a large class of Pure Type Systems (PTSs), including the Calculus of Constructions (CC) without Σ types. They used the standard *double-negation translation*, *i.e.*, the typed variant of Plotkin's original CPS translation, translating source computations of type A to CPS-translated (CPS'd) computations of type $(A^+ \to \bot) \to \bot$ (where $^+$ denotes value translation of types, as explained in Section 3). To avoid certain technical difficulties (which we discuss in Section 5), they consider only *domain-free*

[1]Here "too early" means "before we have a *whole program*"—*i.e.*, before the stage in the compilation pipeline where we link with low-level libraries and code compiled from other languages to form a whole program.
[2]We discuss the popular alternative, ANF translation [Flanagan et al. 1993], in Section 7.

PTSs, a variant of PTSs where λ abstractions do not carry the domain of their bound variable—*i.e.*, they are of the form $\lambda x.\, e$ instead of $\lambda x : A.\, e$ as in the *domain-full* variant.

Barthe and Uustalu [2002] tried to extend these results to the Calculus of Inductive Constructions (CIC), but ended up reporting a negative result, namely that the standard typed CBN CPS translation is *not type preserving* for Σ types. They go on to prove a general impossibility result: for sum types with dependent case analysis, type preservation is *not possible* for the class of CPS translations that can implement `call/cc`. We return to this latter impossibility result, which does not apply to our translation, in Section 7.

The problem of CPS translation for Σ types has been revisited by others over the years, but without positive results. In 2016, Greg Morrisett attempted typed call-by-value (CBV) CPS translation of an A-normalized variant of CC, again using the double-negation translation.[3] He alerted us to the fact that typed CBV CPS translation of Σ seems to work when translating from A-normal form, but the CBV CPS translation of Π types (specifically, the application case) fails to type check.

Why is CPS'ing Dependent Types so Hard? In a dependently typed language like CC or CIC, the power of the type system comes from the ability to express decidable equality between terms and types. Intuitively, these equalities are decided by reducing terms to canonical forms and checking that the resulting *values* are syntactically identical. In the source language—*i.e.*, before CPS translation—since the language is effect-free, every term can be thought of as a value since every term reduces a value. But CPS translation converts source (terms and) values into computations of type $(A \to \bot) \to \bot$. This changes the *interface* to the values—now we can only access the value indirectly, by providing a computation that will do something with the value. In essence, ensuring CPS translations are well typed is hard because every source value has turned into a computation whose "underlying value" isn't directly accessible for purposes of deciding equivalence. In particular, with the double-negation type translation, one cannot recover the underlying value, because every continuation must return false ($\bot$).

This description in terms of interfaces is just a shallow description of the problem. At a deeper level, the problem is that dependently typed languages rely on the ability of the type system to copy expressions from a *term-level* context into a *type-level* context, but CPS transforms expressions into *computations* whose meaning, or "underlying value", depends on its term-level context. This copying happens in particular in the elimination rules for Π and Σ types, and the dependent case analysis of sum types—hence these features are at the heart of past negative results—but in general this happens any time a type depends on an expression. After CPS, we no longer copy an expression, whose meaning is self contained; instead we copy a computation, whose meaning depends on its term-level context. Not only do we "forget" part of the meaning of computations, but as we discussed before, a computation cannot run in a type-level context—it requires a term-level context. As we describe next, our solution to these problems will be to record part of the term-level contexts during type checking and to provide an interface that allows types to run computations.

Answer-Type Polymorphism (and a Free Theorem) to the Rescue! In this paper, we show that type-preserving CBN and CBV CPS translations of CC, with both Σ and Π types, are indeed possible. The key to our result is that we abandon the standard typed CPS translation based on double negation in favor of one that employs *answer-type polymorphism* [Ahmed and Blume 2011; Thielecke 2003, 2004]. Specifically, CPS'd computations are assigned types of the form $\forall\alpha.(A \to \alpha) \to \alpha$.[4] We use answer-type polymorphism because it lets us choose what type of answer we want back from a computation. This gives us the ability to locally "run" any CPS'd computation to extract the

[3] Personal communication, Greg Morrisett, April 2016.

[4] In CC, $\forall\alpha.(A \to \alpha) \to \alpha$ is written $\Pi\alpha : *.(A \to \alpha) \to \alpha$. We switch to the latter after we introduce CC in Section 2.

$$
\begin{array}{llll}
\textit{Universes} & U & ::= & * \mid \Box \\
\textit{Expressions} & t, e, A, B & ::= & * \mid x \mid \Pi\, x : A.\, e \mid \lambda\, x : A.\, e \mid e\ e \mid \mathsf{let}\ x = e : A\ \mathsf{in}\ e \\
 & & \mid & \Sigma\, x : A.\, B \mid \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \\
\textit{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, x = e : A
\end{array}
$$

Fig. 1. CC Syntax

underlying result by running the computation with the identity continuation. We use our ability to extract the underlying result from a computation to recover the equalities needed to prove that CPS translation is type preserving.

We define a CPS target language that extends CC with two additional rules—an equivalence rule and a typing rule—inspired (and justified) by a free theorem for the type $\forall\alpha.(A \to \alpha) \to \alpha$. The two new rules are essential for type checking the previously problematic CBN/CBV CPS translations of Σ and Π types. We show the consistency of our target language by translating it into a parametric model of extensional CC.

Contributions. This paper makes the following contributions.

(1) We give CBN and CBV CPS translations for the *domain-full* version of CC, with Σ as well as Π types, and prove that the translations are type preserving (Section 5 and Section 6). Unlike Barthe et al. [1999], our translation is defined by induction on typing derivations and we avoid the proof-staging difficulties they discussed as their motivation for studying CPS of domain-free CC.
(2) We prove the consistency of our CPS target language CC^k, which includes two additional rules (that, in essence, internalize a free theorem) by showing that we can translate CC^k into a parametric model of extensional CC (Section 4). The translation of the new typing rule resembles the inverse CPS translation of Flanagan et al. [1993].
(3) We prove *separate compilation correctness* of our CPS translations. Since we are in a dependently typed setting, proving type preservation requires proving preservation of reduction, which then easily yields correct separate compilation (Section 5.1 and Section 6.1).
(4) We conjecture that our CPS translation, based on answer-type polymorphism, should work for sum types with dependent elimination and provide a proof sketch (Section 7). We explain why the impossibility proof by Barthe and Uustalu [2002], which applies to a class of CPS translations, does not apply to the answer-type-polymorphism translation and discuss how to tackle other issues we expect in scaling to CIC.

Next, we present our source language CC (Section 2), then discuss cases of the (CBN/CBV) double-negation translation that fail to type check and how we fix the problem (Section 3). Parts of translations and proofs elided from this paper are presented in detail in the online supplementary material [Bowman et al. 2017]. The supplementary material includes both a technical appendix with the additional figures and proofs, and Coq sources for the key lemma in the proof of consistency of our CPS target language CC^k (discussed in Section 4.1).

2 THE CALCULUS OF CONSTRUCTIONS (CC)

Our source language is an extension of the intensional Calculus of Constructions (CC) with strong dependent pairs (Σ types) and dependent let. We typeset this language in a non-bold, blue, sans-serif font. We adapt this presentation from the model of the Calculus of Inductive Constructions (CIC) given in the Coq reference manual [The Coq Development Team 2017, Chapter 4].

We present the syntax of CC in Figure 1 in the style of a Pure Type System (PTS) with no syntactic distinction between terms, which are run-time computations, types, which statically describe terms and compute during type checking, and kinds, which describe types. We use the phrase "expression"

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

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

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

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

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

Fig. 2. CC Convertibility and Equivalence

to refer to a term, type, or kind in the PTS syntax. We usually use the meta-variable e to evoke a term expression and A or B to evoke a type expression. Similarly, we use x to evoke term variables and α for type variables; note that we have no kind-level computation in this language. We use t for an expression to be explicitly ambiguous about its nature as a term, type, or kind.

The language includes one impredicative *universe*, or *sort*, $*$, and its type, $\Box$. The syntax of expressions includes the universe $*$, variables x or α, Π types $\Pi\, x : A.\, B$, functions $\lambda\, x : A.\, e$, application $e_1\ e_2$, dependent let $\mathsf{let}\ x = e : A\ \mathsf{in}\ e'$, Σ types $\Sigma\, x : A.\, B$, dependent pairs $\langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B$, and first and second projections $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$. Note that we cannot write $\Box$ in source programs—it is only used by the type system. The environment Γ includes assumptions $x : A$ and definitions $x = e : A$. Definitions, introduced while type checking let, allow us to convert a variable x to its definition e, called δ-reduction, and provides additional definitional equivalences compared to application.

For brevity, we omit the type annotations on pairs, $\langle e_1, e_2 \rangle$, and let expressions, $\mathsf{let}\ x = e\ \mathsf{in}\ e'$, when they are clear from context. We use the notation $A \to B$ for a function type whose result B does not depend on the input.

In Figure 2 we present the convertibility and equivalence relations for CC. These relations are defined over *untyped* expressions and are used to decide equivalences between types during type checking. The conversion relation can also be seen as the dynamic semantics of programs in CC. It does not fix an evaluation order, but this is not important since CC is effect-free.

We start with the small-step reductions $\Gamma \vdash e \rhd e'$. Note that we label each individual reduction rule with an appropriate subscript, such as $\rhd_\beta$ for β-reduction. When we refer to the undecorated transition $\Gamma \vdash e \rhd e'$ we mean that e reduces to e' using *some* reduction rule— *i.e.*, using one of $\rhd_\delta$, $\rhd_\beta$, $\rhd_\zeta$, $\rhd_{\pi_1}$, or $\rhd_{\pi_2}$. This relation requires the environment Γ for δ-reduction as mentioned previously. For brevity, we usually write this relation as $e \rhd e'$, with the environment Γ as an implicit parameter. This reduction relation is completely standard, although δ-reduction may be surprising to readers unfamiliar with dependent type theory. We can δ-reduce any variable x to its definition e, written $x \rhd_\delta e$.

We define the relation $\Gamma \vdash e \rhd^\star e'$ as the reflexive, transitive, compatible closure of the small-step relation $\Gamma \vdash e \rhd e'$. This relation can apply the small-step relation any number of times to any sub-expression in any order. We usually omit the Γ and write $e \rhd^\star e'$ for brevity, but note that the compatible closure rule for let introduces a new definition into Γ, as follows.

$\boxed{\vdash \Gamma}$

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

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

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \Box}\ [\textsc{Ax-*}] \qquad \frac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ [\textsc{Var}] \qquad \frac{\Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi\, x : A.\, B : *}\ [\textsc{Prod-*}]$$

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

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

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Sigma\, x : A.\, B : *}\ [\textsc{Sigma}] \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B : \Sigma\, x : A.\, B}\ [\textsc{Pair}]$$

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

Fig. 3. CC Typing

$$\frac{\Gamma, x = e : A \vdash e_1 \rhd^\star e_2}{\Gamma \vdash \mathsf{let}\ x = e : A\ \mathsf{in}\ e_1 \rhd^\star \mathsf{let}\ x = e : A\ \mathsf{in}\ e_2}$$

We define definitional equivalence $\Gamma \vdash e \equiv e'$ as reduction in the $\rhd^\star$ relation to the same expression, up to η-equivalence. This algorithmic presentation induces symmetry and transitivity of $\equiv$ without explicit symmetry and transitivity rules, but requires two symmetric versions of η-equivalence. We usually abbreviate this judgment as $e \equiv e'$, leaving Γ implicit.

The typing rules for CC, Figure 3, are completely standard. The judgment $\vdash \Gamma$ checks that the environment Γ is well formed; it is defined by mutual recursion with the typing judgment. The typing judgment $\Gamma \vdash e : A$ checks that expressions are well typed. The rule [Prod-*] implicitly allows impredicativity in $*$, since the domain A could be in the higher universe $\Box$. The rule [Lam] for functions $\lambda\, x : A.\, e$ gives this function the type $\Pi\, x : A.\, B$, binding the function's variable x in the result type B. The rule [App] is the standard dependent application rule. When applying a dependent function $e : \Pi\, x : A.\, B$ to an argument e', the argument is substituted into the result type B yielding an expression $e\ e' : B[e'/x]$. The rule [Let] is similar; however, when checking the body of $\mathsf{let}\ x = e' : A\ \mathsf{in}\ e$, we also adds a definition $x = e' : A$ to the environment. This provides strictly more type expressivity than the application rule, since the body e is typed with respect to a particular value for x while a function is typed with respect to an arbitrary value. The rule [Sigma] ensures we do not allow impredicative strong Σ types, which are inconsistent [Coquand 1986; Hook and Howe 1986]. Note that the type of a dependent pair $\Sigma\, x : A.\, B$ may have the first component x free in the type of the second component B. The rule [Snd] for the second projection of a dependent pair, $\mathsf{snd}\ e$, replaces the free variable x by the first projection, giving $\mathsf{snd}\ e$ the dependent type $B[(\mathsf{fst}\ e)/x]$. Finally, as we have computation in types, the rule [Conv] allows typing an expression $e : A$ as $e : B$ when $A \equiv B$. Note that while the equivalence relation is untyped, we ensure decidability by only using equivalence in [Conv] after type checking both A and B.

$$
\begin{array}{llll}
\textit{Kinds} & \kappa & ::= & * \mid \Pi\, \alpha : \kappa.\, \kappa \mid \Pi\, x : A.\, \kappa \\
\textit{Types} & A, B & ::= & \alpha \mid \lambda\, x : A.\, B \mid \lambda\, \alpha : \kappa.\, B \mid A\ e \mid A\ B \mid \Pi\, x : A.\, B \\
 & & \mid & \Pi\, \alpha : \kappa.\, B \mid \mathsf{let}\ x = e : A\ \mathsf{in}\ B \mid \mathsf{let}\ \alpha = A : \kappa\ \mathsf{in}\ B \mid \Sigma\, x : A.\, B \\
\textit{Terms} & e & ::= & x \mid \lambda\, x : A.\, e \mid \lambda\, \alpha : \kappa.\, e \mid e\ e \mid e\ A \mid \mathsf{let}\ x = e : A\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ \alpha = A : \kappa\ \mathsf{in}\ e \mid \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \\
\textit{Environment} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, x = e : A, \mid \Gamma, \alpha : \kappa \mid \Gamma, \alpha = A : \kappa
\end{array}
$$

Fig. 4. CC Explicit Syntax

To make our upcoming CPS translation easier to follow, we present a second version of the syntax for CC in which we make the distinction between terms, types, and kinds explicit (see Figure 4). The two presentations are equivalent [Barthe et al. 1999]. Distinguishing terms from types and kinds is useful since we only want to CPS translate *terms*, because our goal is to internalize only *run-time* evaluation contexts. (We discuss *pervasive* translation, which also internalizes the *type* contexts, in Section 7.)

3 MAIN IDEAS: HOW DOUBLE-NEGATION CPS FAILS AND HOW TO FIX IT

In Section 1 we informally explained why CPS translation of dependent types causes type preservation to fail. We now make that intuition concrete by studying two examples. We focus on two cases of the double-negation CPS translation that fail to type check: the CBN translation of $\mathsf{snd}\ e$ (reported by Barthe and Uustalu [2002]) and the CBV translation of $e\ e'$ (noticed by Morrisett).

Consider the CBN CPS translation. We translate a term e of type A into a CPS'd computation, written $e^{\div}$, of type $A^{\div}$. Given a type A, we define its *computation translation* $A^{\div}$ and its *value translation* A^{+}. Below, we define the translations for Σ and Π. (Technically, the translations are defined by induction on typing derivations, but we present them less formally in this section.) We write our target language in a **bold, red, serif font**, but for now it is identical to the source language CC.

$$A^{\div} = (A^{+} \to \bot) \to \bot \qquad (\Sigma\, x : A.\, B)^{+} = \mathbf{\Sigma\, x : A^{\div}.\, B^{\div}} \qquad (\Pi\, x : A.\, B)^{+} = \mathbf{\Pi\, x : A^{\div}.\, B^{\div}}$$

Note that since this is the CBN translation, the translated argument type for Π is a computation type $A^{\div}$ instead of a value type A^{+}, and the translated component types for Σ are computation types $A^{\div}$ and $B^{\div}$.

As a warm-up, consider the CBN translation of $\mathsf{fst}\ e$ (where $e : \Sigma\, x : A.\, B$):

$$(\mathsf{fst}\ e : A)^{\div} = \mathbf{\lambda\, k : A^{+} \to \bot.\, e^{\div}\ (\lambda\, y : (\Sigma\, x : A^{\div}.\, B^{\div}).\, let\ z = (fst\ y) : A^{\div}\ in\ z\ k)}$$

It is easy to see that the above type checks (checking the types of $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{k}$).

Next, consider the CBN translation of $\mathsf{snd}\ e$ (where $e : \Sigma\, x : A.\, B$):

$$(\mathsf{snd}\ e : B[\mathsf{fst}\ e/x])^{\div} = \mathbf{\lambda\, k : B^{+}[(fst\ e)^{\div}/x] \to \bot.\, e^{\div}\ (\lambda\, y : (\Sigma\, x : A^{\div}.\, B^{\div}).\, let\ z = (snd\ y) : B^{\div}[fst\ y/x]\ in\ z\ k)}$$

The above does not type check because $\mathbf{z}$ expects a continuation of type $B^{+}[\mathsf{fst}\ \mathbf{y}/\mathbf{x}] \to \bot$ but $\mathbf{k}$ has type $B^{+}[(\mathsf{fst}\ e)^{\div}/\mathbf{x}] \to \bot$. Somehow we need to show that $\mathsf{fst}\ \mathbf{y} \equiv (\mathsf{fst}\ e)^{\div}$. But what is the relationship between $\mathbf{y}$ and e? Intuitively, $e^{\div} : A^{\div}$ is a computation that will pass its result—*i.e.*, the underlying value of type A^{+} inside $e^{\div}$, which corresponds to the value produced by evaluating the source term e—to its continuation. So when $e^{\div}$'s continuation is called, its argument $\mathbf{y}$ will always be equal to the unique "underlying value" inside $e^{\div}$. However, since we have used a *function* to describe a continuation, we must type check the body of the continuation assuming that $\mathbf{y}$ is *any* value of the appropriate type instead of the *exactly one* underlying value from $e^{\div}$.

Even if we could communicate that $\mathbf{y}$ is equal to exactly one value, we have no way to extract the underlying A^{+} value from $e^{\div}$ since the latter takes a continuation that never returns (since it must return a term of type $\bot$). To extract the underlying value from a computation, we need a means of converting from $A^{\div}$ to A^{+}. In essence, after CPS, we have an *interoperability* problem between

the term language (where computations have type $A^{\div}$) and the type language (which needs values of type A^{+}). In the source language, before CPS, we are able to pretend that the term and type languages are the same because all *computations* of type A reduce to *values* of type A. However, the CPS translation creates a gap between the term and type languages; it changes the *interface* to terms so that the only way to get a value out of a computation is to have a continuation, which can never return, ready to receive that value.

The solution to both of the above problems requires a CPS translation based on answer-type polymorphism. That is, we change the computation translation to $A^{\div} = \Pi\, \alpha : *.\, (A^{+} \to \alpha) \to \alpha$. Now, to extract the underlying A^{+} value from $e^{\div} : A^{\div}$, we can run $e^{\div}$ with the identity continuation as follows: $e^{\div}\ A^{+}\ \mathbf{id}$. Moreover, we can now justify type checking the body of $e^{\div}$'s continuation under the assumption that $\mathbf{y} \equiv e^{\div}\ A^{+}\ \mathbf{id}$ thanks to a *free theorem* we get from the type $A^{\div}$. The free theorem says that running some $\mathbf{e} : A^{\div}$ with continuation $\mathbf{k} : \mathbf{A} \to \mathbf{B}$ is equivalent to running $\mathbf{e}$ with the identity continuation and then passing the result to $\mathbf{k}$, *i.e.*, $\mathbf{e\ B\ k} \equiv \mathbf{k}\ (\mathbf{e\ A\ id})$.

To formalize this intuition in our target language, we first add new syntax for the application of a computation to its answer type and continuation: $\mathbf{e\ @\ A\ e'}$. Next, we internalize the aforementioned free theorem by adding two rules to our target language. The first is the following typing rule which records (a representation of) the *value* of a computation while type checking a continuation. That is, it allows us to assume $\mathbf{y} \equiv e^{\div}\ A^{+}\ \mathbf{id}$ when type checking the body of $e^{\div}$'s continuation.

$$\frac{\Gamma \vdash \mathbf{e} : \Pi\, \alpha : *.\, (\mathbf{A} \to \alpha) \to \alpha \qquad \Gamma \vdash \mathbf{B} : * \qquad \Gamma, \mathbf{x} = \mathbf{e\ A\ id} \vdash \mathbf{e'} : \mathbf{B}}{\Gamma \vdash \mathbf{e\ @\ B}\ (\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{e'}) : \mathbf{B}}\ [\text{T-Cont}]$$

The second is the following equivalence rule, which is justified by the free theorem. Intuitively, this rule normalizes CPS'd computations to the "value" $e^{\div}\ A^{+}\ \mathbf{id}$.

$$\frac{}{\Gamma \vdash (\mathbf{e}_1\ \mathbf{@\ B}\ (\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{e}_2)) \equiv (\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{e}_2)\ (\mathbf{e}_1\ \mathbf{A\ id})}\ [\equiv\text{-Cont}]$$

Here is the updated CPS translation $(\mathsf{snd}\ e : B[\mathsf{fst}\ e/x])^{\div}$ that leverages answer-type polymorphism:

$$\lambda\, \alpha : *.\, \lambda\, \mathbf{k} : B^{+}[(\mathsf{fst}\ e)^{\div}/\mathbf{x}] \to \alpha.\, e^{\div}\ \mathbf{@}\ \alpha\ (\lambda\, \mathbf{y} : (\Sigma\, \mathbf{x} : A^{\div}.\, B^{\div}).\, \mathbf{let\ z =\ (snd\ y)} : B^{\div}[\mathbf{fst\ y}/\mathbf{x}]\ \mathbf{in\ z}\ \alpha\ \mathbf{k})$$

To type check $e^{\div}\ \mathbf{@}\ \alpha\ \ldots$ we use [T-Cont]. When type checking the body of $e^{\div}$'s continuation, we have that $\mathbf{y} \equiv e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}.\, B^{\div})\ \mathbf{id}$ and recall that we need to show that $\mathbf{fst\ y} \equiv (\mathsf{fst}\ e)^{\div}$. This requires expanding $(\mathsf{fst}\ e)^{\div}$ and making use of the [≡-Cont] rule we now have available in the target language. Here is an informal sketch of the proof—we give the detailed proof in Section 5.1.

$$\begin{aligned}
(\mathsf{fst}\ e)^{\div} &\equiv e^{\div}\ \mathbf{@}\ \alpha'\ (\lambda\, \mathbf{y}.\, \mathbf{fst\ y}) && \text{by (roughly) the translation of fst and by } \eta\text{-equivalence} && (1)\\
&\equiv (\lambda\, \mathbf{y}.\, \mathbf{fst\ y})\ (e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}.\, B^{\div})\ \mathbf{id}) && \text{by [≡-Cont]} && (2)\\
&\equiv \mathbf{fst}\ (e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}.\, B^{\div})\ \mathbf{id}) && \text{by reduction} && (3)\\
&\equiv \mathbf{fst\ y} && \text{by } \mathbf{y} \equiv e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}.\, B^{\div})\ \mathbf{id} && (4)
\end{aligned}$$

The astute reader will have noticed that our CPS translation—as well as the new rules [T-Cont] and [≡-Cont]—only uses the new @ syntax for certain applications. Intuitively, we use @ only when type checking requires the free theorem.

Next, let's look at the translation of Π types. Again, we start with a warm-up; consider the following CBN double-negation CPS translation of $e\ e'$ (where $e : \Pi\, x : A.\, B$ and $e' : A$):

$$(e\ e' : B[e'/x])^{\div} = \lambda\, \mathbf{k} : (B^{+}[e'^{\div}/\mathbf{x}]) \to \bot.\, e^{\div}\ (\lambda\, \mathbf{f} : \Pi\, \mathbf{x} : A^{\div}.\, B^{\div}.\, (\mathbf{f}\ e'^{\div})\ \mathbf{k})$$

The above type checks (as seen by inspecting the types of $\mathbf{f}$ and $\mathbf{k}$). Notice that $e'^{\div}$ appears as an argument to $\mathbf{f}$ so the type of $\mathbf{f}\ e'^{\div} : B^{\div}[e'^{\div}/\mathbf{x}]$.

Now consider the CBV CPS translation based on double negation, which fails to type check. We define the CBV *computation translation* $A^{\div}$ and *value translation* A^{+} as follows.

$$A^\div = (A^+ \to \bot) \to \bot \qquad (\Sigma\, x : A.\, B)^+ = \Sigma\, \mathbf{x} : A^+.\, B^+ \qquad (\Pi\, x : A.\, B)^+ = \Pi\, \mathbf{x} : A^+.\, B^\div$$

Since we're doing a CBV translation, the translated argument Π is a value of type A^+ and the translated component types for Σ are values of types A^+ and B^+.

Here is the CBV CPS translation $e\ e'$ (where $e : \Pi\, x : A.\, B$ and $e' : A$):

$$(e\ e' : B[e'/x])^\div = \lambda\, \mathbf{k} : (B^+[e'^+/\mathbf{x}]) \to \bot.\, e^\div\ (\lambda\, \mathbf{f} : \Pi\, \mathbf{x} : A^+.\, B^\div.\, e'^\div\ (\lambda\, \mathbf{x} : A^+.\, (\mathbf{f}\ \mathbf{x})\ \mathbf{k}))$$

For the moment, ignore that our type annotation on $\mathbf{k}$, $(B^+[e'^+/\mathbf{x}])$, seems to requires a value translation of *terms* e'^+, which we can't normally define. Instead, notice that unlike in the CBN translation, we now evaluate the argument $e'^\div$ before calling $\mathbf{f}$, so in CBV we have the application $\mathbf{f}\ \mathbf{x} : B^\div[\mathbf{x}/\mathbf{x}]$. This translation fails to type check since the computation $\mathbf{f}\ \mathbf{x}$ expects a continuation of type $(B^+[\mathbf{x}/\mathbf{x}]) \to \bot$ but $\mathbf{k}$ has type $(B^+[e'^+/\mathbf{x}]) \to \bot$. Somehow we need to show that $\mathbf{x} \equiv e'^+$. This situation is almost identical to what we saw with the failing CBN translation of snd e. Analogously, this time we ask what is the relationship between $\mathbf{x}$ and $e'^\div$, or e'^+? As before, we note that the only value that can flow into $\mathbf{x}$ is the unique underlying value in $e'^\div$.

Hence, fortunately, the solution is again to do what we did for the CBN translation: adopt a CPS translation based on answer-type polymorphism. As before, we change the computation translation to $A^\div = \Pi\, \alpha : *.\, (A^+ \to \alpha) \to \alpha$. Here is the updated CBV CPS translation of $(e\ e' : B[e'/x])^\div$:

$$\lambda\, \alpha : *.\, \lambda\, \mathbf{k} : (B^+[(e'^\div\ A^+\ \mathbf{id})/\mathbf{x}]) \to \alpha.\, e^\div\ \alpha\ (\lambda\, \mathbf{f} : \Pi\, \mathbf{x} : A^+.\, B^\div.\, e'^\div\ @\ \alpha\ (\lambda\, \mathbf{x} : A^+.\, (\mathbf{f}\ \mathbf{x})\ \alpha\ \mathbf{k}))$$

First, notice that we use the new @ form when evaluating the argument $e'^\div$, which tells us we're using our new typing rule to record the *value* of $e'^\div$ while we type check its continuation. Second, notice the type annotation on $\mathbf{k}$. Earlier we observed that the type annotation for $\mathbf{k}$, $(B^+[e'^+/\mathbf{x}])$, seemed to require a value translation on terms e'^+ that cannot normally be defined. Our translation gives us a sensible way of modeling the value translation of a term by invoking a computation with the identity continuation—so e'^+ is just the underlying value in $e'^\div$, *i.e.*, $(e'^\div\ A^+\ \mathbf{id})$. This is an important point to note: unlike CBN CPS, where we can substitute computations for variables, in CBV CPS we must find a way to extract the underlying value from computations of type $A^\div$ since variables expect values of type A^+. Without answer-type polymorphism, CBV CPS is, in some sense, much more broken than CBN CPS! Indeed, Barthe et al. [1999] already gave a CBN double-negation CPS translation for CC's Π types, but typed CBV double-negation CPS of Π types fails.

Using the new typing rule and equivalence rule that we already added to our target languages, we are able to type check the above translation of $e\ e'$ in essentially the same way as we did for the CBN translation of snd e. We show the detailed proof in Section 6.1.

The reader may worry that our CBV CPS translation produces many terms of the form $\mathbf{k}\ (e^\div\ A^+\ \mathbf{id})$, which aren't really in CPS since $e^\div\ A^+\ \mathbf{id}$ must return. However, notice that these only appear in type annotations, not as run-time expressions. We only run a computation with the identity continuation to convert a CPS expression into a value *in the types* for deciding type equivalence. The run-time terms are all in CPS and can be run in a machine-like semantics in which computations do not return.

4 THE CALCULUS OF CONSTRUCTIONS WITH CPS AXIOMS (CC^K)

Our target language CC^k is CC extended with a syntax for parametric reasoning about computations in CPS form, as discussed in Section 3. We present these extensions formally in Figure 5. We add the form $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ to the syntax of CC^k. This form represents a computation $\mathbf{e}$ applied to the answer type $\mathbf{A}$ and the continuation $\mathbf{e}'$. The dynamic semantics are the same as standard application. The equivalence rule [≡-Cont] states that a computation $\mathbf{e}$ applied to its continuation $\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}'$ is equivalent to the application of that continuation to the underlying value of $\mathbf{e}$. We extract the

underlying value by applying $\mathbf{e}$ to the "halt continuation", encoded as the identity function in our system. The rule [T-Cont] is used to type check applications that use our new @ syntax. This typing rule internalizes the fact that a continuation will be applied to one particular input, rather than an arbitrary value. It tells the type system that the application of a computation to a continuation $\mathbf{e}\ @\ \mathbf{A}\ (\lambda\, \mathbf{x} : \mathbf{B}.\ \mathbf{e}')$ jumps to the continuation $\mathbf{e}'$ after evaluating $\mathbf{e}$ to a value and binding the result to $\mathbf{x}$. We check the body of the continuation $\mathbf{e}'$ under the assumption that $\mathbf{x} = \mathbf{e}\ \mathbf{B}\ \mathbf{id}$, *i.e.*, with the equality that the name $\mathbf{x}$ refers to the underlying value in the computation $\mathbf{e}$, which we access using the interface given by the polymorphic answer type.

The rule [≡-Cont] is a declarative rule that requires explicit symmetry and transitivity rules to complete the definition (elided here, but included in the supplementary material). We give a declarative presentation of this rule for clarity.

Note that [≡-Cont] and [T-Cont] internalize a specific "free theorem" that we need to prove type preservation of the CPS translation. In particular, [≡-Cont] only holds when the CPS'd term $\mathbf{e}_1$ has the expected parametric type $\Pi\, \alpha : *.\ (\mathbf{A} \to \alpha) \to \alpha$ given in [T-Cont]. Notice, however, that our statement of [≡-Cont] does not put any requirements on the type of $\mathbf{e}_1$. This is because we use an untyped equivalence based on the presentation of CIC in Coq [The Coq Development Team 2017, Chapter 4], and this untyped equivalence is necessary in our type-preservation proof (see Section 5.1). Therefore, we cannot simply add typing assumptions directly to [≡-Cont]. Instead, we rely on the fact that the term $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ has only one introduction rule, [T-Cont]. Since there is only one applicable typing rule, anytime $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ appears in our type system, $\mathbf{e}$ has the required parametric type. Furthermore, while our equivalence is untyped, we never appeal to equivalence with ill-typed terms; we only refer to the equivalence $\mathbf{A}' \equiv \mathbf{B}'$ in [Conv] *after* checking that both $\mathbf{A}'$ and $\mathbf{B}'$ are well-typed. For example, suppose the term $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ occurs in type $\mathbf{A}'$, and to prove that $\mathbf{A}' \equiv \mathbf{B}'$ requires our new rule [≡-Cont]. Because $\mathbf{A}'$ is well-typed, we know that its subterms, including $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$, are well-typed. Since $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ can only be well-typed by [T-Cont], we know $\mathbf{e}$ has the required parametric type.

Finally, notice that in [T-Cont] and [≡-Cont] we use standard application syntax for the term $\mathbf{e}\ \mathbf{B}\ \mathbf{id}$. We only use the @ syntax in our CPS translation when we require one of our new rules. The type of the identity function doesn't depend on any value, so we never need [T-Cont] to type-check the identity continuation. In a sense, $\mathbf{e}\ \mathbf{B}\ \mathbf{id}$ is the normal form of a CPS'd "value" so we never need [≡-Cont] to rewrite this term—*i.e.*, using [≡-Cont] to rewrite $\mathbf{e}\ \mathbf{B}\ \mathbf{id}$ to $\mathbf{id}\ (\mathbf{e}\ \mathbf{B}\ \mathbf{id})$ would just evaluate the original term.

4.1 Consistency of CC^k

We prove that CC^k is consistent by giving a model of CC^k in the extensional Calculus of Constructions. Boulier et al. [2017] provide a detailed explanation of this standard technique.

The idea behind the model is that we can translate each use of [≡-Cont] in CC^k to a propositional equivalence in extensional CC. Next, we translate any term that is typed by [T-Cont] into a dependent let. Finally, we establish that if there were a proof of $\mathbf{False}$ in CC^k, our translation would construct a proof of *False* in extensional CC. But since extensional CC is consistent, there can be no proof of $\mathbf{False}$ in CC^k. We construct the model in three parts.

(1) produce proofs (in extensional CC) of all propositional equivalences introduced by our translation of [≡-Cont],
(2) show $\mathbf{False}$ in CC^k is translated to *False* in extensional CC,
(3) show our translation is type preserving, *i.e.*, it translates a proof of $\mathbf{A}$ to a proof of its translation $\mathbf{A}^\circ$.

Extensions to Syntax, Figure 1

$$\mathbf{e} \quad ::= \quad \cdots \mid \mathbf{e} \ @ \ \mathbf{A} \ \mathbf{e}'$$

Extensions to Convertibility and Equivalence, Figure 2

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

$$\cdots$$
$$\lambda\, \alpha : *.\, \mathbf{e}_1 \ @ \ \mathbf{A}\ \mathbf{e}_2 \quad \rhd_@ \quad (\mathbf{e}_1[\mathbf{A}/\alpha])\ \mathbf{e}_2$$

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

$$\ldots \qquad \frac{}{\Gamma \vdash (\mathbf{e}_1 \ @ \ \mathbf{A}\ (\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}_2)) \equiv (\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}_2)\ (\mathbf{e}_1\ \mathbf{B}\ \mathbf{id})}\ [\equiv\text{-Cont}]$$

Extensions to Typing, Figure 3

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

$$\ldots \qquad \frac{\Gamma \vdash \mathbf{e} : \Pi\, \alpha : *.\, (\mathbf{B} \to \alpha) \to \alpha \qquad \Gamma \vdash \mathbf{A} : * \qquad \Gamma, \mathbf{x} = \mathbf{e}\ \mathbf{B}\ \mathbf{id} \vdash \mathbf{e}' : \mathbf{A}}{\Gamma \vdash \mathbf{e} \ @ \ \mathbf{A}\ (\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}') : \mathbf{A}}\ [\text{T-Cont}]$$

Fig. 5. CC^k: CC with extensions

$\boxed{\Gamma \vdash e_1 \equiv e_2}$

$$\ldots \qquad \frac{\Gamma \vdash p : e_1 = e_2}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Ext}]$$

Fig. 6. Additional Equivalence Rule for Extensional CC

As our model is in the extensional CC, it is not clear that type checking in CC^k is decidable. We believe that type checking should be decidable for all programs produced by our compiler, since type checking in our source language CC is decidable. In the worst case, to ensure decidability we could change our translation to use a propositional version of [≡-Cont]. The definitional presentation is simpler, but it should be possible to change the translation so that, in any term that currently relies on [≡-Cont], we insert type annotations that compute type equivalence using a propositional version of [≡-Cont]. We leave the issue of decidability of type checking in CC^k for future work.

4.1.1 Modeling [≡-Cont]. The extensional Calculus of Constructions differs from our source language CC in only one way: it allows using the existence of a propositional equivalence as a definitional equivalence, as shown in Figure 6. The syntax and typing rules are exactly the same as in CC presented in Section 2. We write terms in extensional CC using a *italic, black, serif font.*

In extensional CC we can model each use of the definitional equivalence [≡-Cont] by [≡-Ext], as long as there exists a proof $p : (e\ A\ k) = (k\ (e\ B\ id))$, *i.e.*, a propositional proof of [≡-Cont]; we prove this propositional proof always exists by using the parametricity translation of Keller and Lasson [2012]. This translation gives a parametric model of CC in itself. This translation is based on prior translations that apply to all Pure Type Systems [Bernardy et al. 2012], but includes an impredicative universe and provides a Coq implementation that we use.

The translation of a type A, written $[\![A]\!]$, essentially transforms the type into a relation on terms of that type. On terms e of type A, the translation $[\![e]\!]$ produces a proof that e is related to itself in the relation given by $[\![A]\!]$. For example, a type $*$ is translated to the relation $[\![*]\!] = \lambda\, (x, x' : *).\, x \to x' \to *$. The translation of a polymorphic function type $[\![\Pi\, \alpha : *.\, A]\!]$ is the following.

$$\lambda\, (f, f' : (\Pi\, \alpha : *.\, A)).\, \Pi\, (\alpha, \alpha' : *).\, \Pi\, \alpha_r : [\![*]\!]\ \alpha\ \alpha'.\, ([\![A]\!]\ (f\ \alpha)\ (f'\ \alpha'))$$

This relation produces a proof that the bodies of functions f and f' are related when provided a relation α_r for the two types of α and α'. This captures the idea that functions at this type must

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

$$\ldots \quad \frac{\Gamma \vdash \mathbf{e} : \_ \rightsquigarrow_\circ e \qquad \Gamma \vdash \mathbf{B} : \_ \rightsquigarrow_\circ B \qquad \Gamma \vdash \mathbf{A} : \_ \rightsquigarrow_\circ A \qquad \Gamma, \mathbf{x} = \mathbf{e}\ \mathbf{B}\ \mathbf{id} \vdash \mathbf{e'} : \mathbf{A} \rightsquigarrow_\circ e'}{\Gamma \vdash \mathbf{e}\ @\ \mathbf{A}\ (\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{B}.\, \mathbf{e'}) : \mathbf{A} \rightsquigarrow_\circ \mathit{let}\, x = e\ B\ \mathit{id} : B\ \mathit{in}\ e'} \text{ [Un-Cont]}$$

Fig. 7. Translation from CC^k to Extensional CC (excerpt)

behave parametrically in the abstract type α. This translation gives us Theorem 4.1 (Parametricity for extensional CC), *i.e.*, that every expression in extensional CC is related to itself in the relation given by its type.

Theorem 4.1 (Parametricity for extensional CC). *If* $\Gamma \vdash t : t'$ *then* $[\![\Gamma]\!] \vdash [\![t]\!] : [\![t']\!]\ t\ t$

We apply Theorem 4.1 to our CPS type $\Pi\, \alpha : *.\, (B \to \alpha) \to \alpha$ to prove Lemma 4.2. Since a CPS'd term is a polymorphic function, we get to provide a relation α_r for the type α. The translation then gives us a proof that $e\ A\ k$ and $e\ B\ id$ are related by α_r, so we simply choose α_r to be a relation that guarantees $e\ A\ k = k\ (e\ B\ id)$. We formalize part of the proof in Coq in our supplementary materials [Bowman et al. 2017]. By [≡-Ext], Theorem 4.1, and the relation just described, we arrive at a proof of Lemma 4.2 for CPS'd computations encoded in the extensional CC.

Lemma 4.2 (Continuation Shuffling). *If* $\Gamma \vdash A : *$, $\Gamma \vdash B : *$, $\Gamma \vdash e : \Pi\, \alpha : *.\, (B \to \alpha) \to \alpha$, $\Gamma \vdash k : B \to A$, *and* $\Gamma \vdash e : \Pi\, \alpha : *.\, (B \to \alpha) \to \alpha$ *then* $\Gamma \vdash e\ A\ k \equiv k\ (e\ B\ id)$

Note that this lemma relies on the type of the term e. We must only appeal to this lemma, and the equivalence justified by it, when e has the right type. In CC^k, this is guaranteed by the typing rule [T-Cont], as discussed earlier in this section.

4.1.2 Modeling [T-Cont]. In Figure 7 we present the key translation rule for modeling CC^k in extensional CC. All other rules are inductive on the structure of typing derivations. Note that since we only need to justify the additional typing rule [T-Cont], this is the only rule that is changed by the translation. This translation rule is essentially the same rule from the inverse CPS given by Flanagan et al. [1993], although we do not necessarily produce output in A-normal form (ANF) since we only translate uses of this one typing rule. We discuss CPS and ANF in detail in Section 7.

For brevity in our proofs, we define the following notation for the translation of terms and types from CC^k into extensional CC.

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

By writing $\mathbf{e}^\circ$, we refer to the term produced by the translation with the typing derivation $\Gamma \vdash \mathbf{e} : \mathbf{A}$ as an implicit parameter.

First, we show that the definition of False is preserved. We define False as $\Pi\, \boldsymbol{\alpha} : *.\, \boldsymbol{\alpha}$, *i.e.*, the function that accepts any proposition and returns a proof that the proposition holds. It is simple to see that this type has type $*$ in CC^k by the rule [Prod-*]. Note that $\Pi\, \boldsymbol{\alpha} : *.\, \boldsymbol{\alpha}$ is translated to $\Pi\, \alpha : *.\, \alpha$ of type $*$, *i.e.*, False is translated to *False*.

Lemma 4.3 (False Preservation). $\Gamma \vdash (\Pi\, \boldsymbol{\alpha} : *.\, \boldsymbol{\alpha}) : * \rightsquigarrow_\circ \Pi\, \alpha : *.\, \alpha$

Next, to show type preservation, we must first show that equivalence is preserved since the type system appeals to equivalence. A crucial lemma to both equivalence preservation and type preservation is *compositionality*, which says that the translation commutes with substitution. The proof is straightforward by induction on the typing derivation of $\mathbf{e}$. See our technical appendix for details [Bowman et al. 2017].

Lemma 4.4 (Compositionality). $(\mathbf{e}[\mathbf{e'}/\mathbf{x}])^\circ \equiv \mathbf{e}^\circ[\mathbf{e'}^\circ/x]$

The equivalence rules of extensional CC with the addition of Lemma 4.2 (Continuation Shuffling) are the same as CC^k. Therefore, to show that equivalence is preserved, it suffices to show that reduction sequences are preserved. We first show that single-step reduction is preserved, Lemma 4.5, which easily implies preservation of reduction sequences, Lemma 4.6.

Lemma 4.5 (Preservation of One-Step Reduction). *If* $e_1 \triangleright e_2$, *then* $e_1^\circ \triangleright^\star e'$ *and* $e_2^\circ \equiv e'$

Proof. By cases on the reduction step $e_1 \triangleright e_2$. There is one interesting case.

Case $e = (\lambda\, \alpha : *.\, e_1)\ @\ B\ (\lambda\, x' : A.\, e_2) \triangleright_@ (e_1[B/\alpha])\ (\lambda\, x' : A.\, e_2)$

By definition $e^\circ = (let\ x' = ((\lambda\, \alpha : *.\, e_1)^\circ\ A^\circ\ id)\ in\ e_2^\circ) \triangleright_\zeta e_2^\circ[((\lambda\, \alpha : *.\, e_1)^\circ\ A^\circ\ id)/x']$

We must show $((e_1[B/\alpha])\ (\lambda\, x' : A.\, e_2))^\circ \equiv e_2^\circ[((\lambda\, \alpha : *.\, e_1)^\circ\ A^\circ\ id)/x']$.

$$
\begin{aligned}
&((e_1[B/\alpha])\ (\lambda\, x' : A.\, e_2))^\circ && && (5)\\
&\equiv ((e_1^\circ[B^\circ/\alpha])\ (\lambda\, x' : A^\circ.\, e_2^\circ)) && \text{by Lemma 4.4 and definition of } ^\circ && (6)\\
&\equiv (\lambda\, \alpha : *.\, e_1^\circ)\ B^\circ\ (\lambda\, x' : A^\circ.\, e_2^\circ) && \text{by } [\equiv] \text{ and } \triangleright_\beta && (7)\\
&\equiv (\lambda\, x' : A^\circ.\, e_2^\circ)\ ((\lambda\, \alpha : *.\, e_1^\circ)\ A^\circ\ id) && \text{by Lemma 4.2 (Continuation Shuffling)} && (8)\\
&\equiv e_2^\circ[((\lambda\, \alpha : *.\, e_1^\circ)\ A^\circ\ id)/x'] && \text{by } [\equiv] \text{ and } \triangleright_\beta && (9)\\
&\equiv e_2^\circ[((\lambda\, \alpha : *.\, e_1)^\circ\ A^\circ\ id)/x'] && \text{by Lemma 4.4} && (10)
\end{aligned}
$$

□

Lemma 4.6 (Preservation of Reduction Sequences). *If* $e_1 \triangleright^\star e_2$, *then* $e_1^\circ \triangleright^\star e'$ *and* $e_2^\circ \equiv e'$.

Lemma 4.7 (Equivalence Preservation). *If* $e_1 \equiv e_2$, *then* $e_1^\circ \equiv e_2^\circ$

Finally, we can show type preservation, which completes our proof of consistency. Since the translation is homomorphic on all typing rules except [T-Cont], there is only one interesting case in the proof of Lemma 4.8. We must show that [Un-Cont] is type preserving. Note that the case for [Conv] appeals to Lemma 4.7.

Lemma 4.8 (Type Preservation). *If* $\Gamma \vdash e : A$ *then* $\Gamma^\circ \vdash e^\circ : A^\circ$

Proof. By induction on the derivation $\Gamma \vdash e : A$. There is one interesting case.

Case [T-Cont]

We have the following.

$$\frac{\Gamma \vdash e_1 : \Pi\, \alpha : *.\, (B \to \alpha) \to \alpha \qquad \Gamma \vdash A : * \qquad \Gamma, x' = e_1\ B\ id \vdash e_2 : A}{\Gamma \vdash e_1\ @\ A\ (\lambda\, x' : B.\, e_2) : A}$$

We must show $\Gamma^\circ \vdash let\ x' = (e_1^\circ\ B^\circ\ id)\ in\ e_2^\circ : A^\circ$.

By [Let], it suffices to show

- $\Gamma^\circ \vdash (e_1^\circ\ B^\circ\ id) : B^\circ$, which follows easily by the induction hypothesis applied to the premises of [T-Cont].
- $\Gamma^\circ, x' = (e_1^\circ\ B^\circ\ id) : B^\circ \vdash e_2^\circ : A^\circ$, which follows immediately by the induction hypothesis. □

Theorem 4.9 (Consistency of CC^k). *There does not exist a closed term* e *such that* $\cdot \vdash e : False$.

5 CALL-BY-NAME CPS TRANSLATION OF CC

We now present our call-by-name CPS translation (CPS^n) of CC. The main differences between our translation and the one by Barthe and Uustalu [2002] are that we use a locally polymorphic answer type instead of a fixed answer type, which enables our type-preservation proof of snd e, and that we use a domain-full target language, which supports decidable type-checking.

As we discussed in Section 3, we need a computation translation and a value translation on types. But in addition to types, we will need to translate universes, kinds, and terms as well. All of our translations are defined by induction on the typing derivations. This is important when translating to a domain-full target language, since the domain annotations we generate come from the type of the term we are translating. However, as in Section 3, we find it useful to abbreviate these with $t^{\div}$ (for computation) and t^{+} (for value translation). Below we give abbreviations for all of the translation judgments we define for our CPS translation. Note that anywhere we use this notation, we require the typing derivation as an implicit parameter.

$$A^{\div} \stackrel{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash A : * \leadsto^{n}_{A^{\div}} \mathbf{A} \qquad U^{+} \stackrel{\text{def}}{=} \mathbf{U} \text{ where } \Gamma \vdash U \leadsto^{n}_{U} \mathbf{U}$$

$$e^{\div} \stackrel{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash e : A \leadsto^{n}_{e} \mathbf{e} \qquad \kappa^{+} \stackrel{\text{def}}{=} \boldsymbol{\kappa} \text{ where } \Gamma \vdash \kappa : U \leadsto^{n}_{\kappa} \boldsymbol{\kappa}$$

$$A^{+} \stackrel{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash A : \kappa \leadsto^{n}_{A} \mathbf{A}$$

The CPS^n translations on universes, kinds, and types are defined in Figure 8. We define the translation for kinds CPS^n_κ and universe CPS^n_U, which we abbreviate with $^+$. There is no separate computation translation for kinds or universes. We only have separate computation and value translations for *types* since we only internalize the concept of evaluation at the *term*-level, and *types* describe term-level computations and term-level values. Recall that this is the call-by-name translation, so function arguments, even type-level functions, are still computations. Note, therefore, that the rule [CPS^n_κ-ProdA] uses the computation translation on the domain annotation A of $\Pi\, x{:}A.\, \kappa$—*i.e.*, the kind describing a type-level function that abstracts over a term of type A.

For types, we define a value translation CPS^n_A and a computation translation $\text{CPS}^n_{A^{\div}}$. Most rules are straightforward. We translate type-level variables α in-place in rule [CPS^n_A-Var]. Again, since this is the CBN translation, we use the computation translation on domain annotations. The rule [CPS^n_A-Constr] for the value translation of type-level functions that abstract over a term, $\lambda\, x{:}A.\, B$, translates the domain annotation A using the computation translation. The rule for the value translation of a function type, [CPS^n_A-Prod], translates the domain annotation A using the computation translation. This means that a function is a value when it accepts a computation as an argument. The rule [CPS^n_A-Sigma] produces the value translation of a pair type by translating both components of a pair using the computation translation. This means we consider a pair a value when it contains computations as components. Note that since our translation is defined on typing derivations, we have an explicit translation of the conversion rule [CPS^n_A-Conv].

There is only one rule for the computation translation of a type, [$\text{CPS}^n_{A^{\div}}$-Comp], which is the polymorphic answer type translation described in Section 3. Notice that [$\text{CPS}^n_{A^{\div}}$-Comp] is defined only for types of kind $*$, since only types of kind $*$ have inhabitants. For example, we cannot apply [$\text{CPS}^n_{A^{\div}}$-Comp] to type-level function since no term inhabits a type-level function.

The CPS^n translation on terms is defined in Figure 9. Intuitively, we translate each term e of type A to $\mathbf{e}$ of type $\mathbf{\Pi\, \alpha : *.\, (A \to \alpha) \to \alpha}$, where $\mathbf{A}$ is the *value translation* of A. This type represents a computation that, when given a continuation $\mathbf{k}$ that expects a value of type $\mathbf{A}$, promises to call $\mathbf{k}$ with a value of type $\mathbf{A}$. Since we have only two value forms in the call-by-name translation, we do not explicitly define a separate value translation, but inline that translation. Note that the value cases, [CPS^n_e-Fun] and [CPS^n_e-Pair], feature the same pattern: produce a computation $\boldsymbol{\lambda\, \alpha.\, \lambda\, k.\, k\ v}$ that expects a continuation and then immediately calls that continuation on the value $\mathbf{v}$. In the case of [CPS^n_e-Fun], the value $\mathbf{v}$ is the function $\boldsymbol{\lambda\, x : A.\, e}$ produced by translating the source function $\lambda\, x{:}A.\, e$ using the computation type translation from $A \leadsto^n_{A^{\div}} \mathbf{A}$ and the computation term translation $e \leadsto^n_e \mathbf{e}$. In the case of [$\text{CPS}^n_e$-Pair], the value we produce $\langle \mathbf{e}_1, \mathbf{e}_2 \rangle$ contains computations, not values.

The rest of the translation rules are for computations. Notice that while all terms produced by the term translation have a computation type, all continuations take a value type. Since this is a CBN translation, we consider variables as computations in [CPS^n_e-Var]. We translate term variables

$\boxed{\Gamma \vdash U \rightsquigarrow^n_U \mathbf{U}}$

$$\frac{}{\Gamma \vdash * \rightsquigarrow^n_U *}\ [\mathrm{CPS}^n_U\text{-Star}] \qquad \frac{}{\Gamma \vdash \Box \rightsquigarrow^n_U \Box}\ [\mathrm{CPS}^n_U\text{-Box}]$$

$\boxed{\Gamma \vdash \kappa : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}$ Lemma 5.5 will show $\Gamma^+ \vdash \kappa^+ : U^+$

$$\frac{}{\Gamma \vdash * : \Box \rightsquigarrow^n_\kappa *}\ [\mathrm{CPS}^n_\kappa\text{-Ax}] \qquad \frac{\Gamma \vdash \kappa : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : \kappa \vdash \kappa' : U' \rightsquigarrow^n_\kappa \boldsymbol{\kappa'}}{\Gamma \vdash \Pi\, \alpha : \kappa.\, \kappa' : U' \rightsquigarrow^n_\kappa \mathbf{\Pi}\, \boldsymbol{\alpha} : \boldsymbol{\kappa}.\, \boldsymbol{\kappa'}}\ [\mathrm{CPS}^n_\kappa\text{-ProdK}]$$

$$\frac{\Gamma \vdash A : \kappa' \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x : A \vdash \kappa : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\Gamma \vdash \Pi\, x : A.\, \kappa : U \rightsquigarrow^n_\kappa \mathbf{\Pi}\, \mathbf{x} : \mathbf{A}.\, \boldsymbol{\kappa}}\ [\mathrm{CPS}^n_\kappa\text{-ProdA}]$$

$\boxed{\Gamma \vdash A : \kappa \rightsquigarrow^n_A \mathbf{A}}$ Lemma 5.5 will show $\Gamma^+ \vdash A^+ : \kappa^+$

$$\ldots \qquad \frac{}{\Gamma \vdash \alpha : \kappa \rightsquigarrow^n_A \boldsymbol{\alpha}}\ [\mathrm{CPS}^n_A\text{-Var}] \qquad \frac{\Gamma \vdash A : \kappa' \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa \rightsquigarrow^n_A \mathbf{B}}{\Gamma \vdash \lambda\, x : A.\, B : \Pi\, x : A.\, \kappa \rightsquigarrow^n_A \boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ [\mathrm{CPS}^n_A\text{-Constr}]$$

$$\frac{\Gamma \vdash A : \Pi\, x : B.\, \kappa \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash e : B \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash A\, e : \kappa[e/x] \rightsquigarrow^n_A \mathbf{A}\, \mathbf{e}}\ [\mathrm{CPS}^n_A\text{-AppConstr}]$$

$$\frac{\Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa' \rightsquigarrow^n_{A^\div} \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \kappa' \rightsquigarrow^n_A \mathbf{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ [\mathrm{CPS}^n_A\text{-Prod}]$$

$$\frac{\Gamma \vdash \kappa : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, x : A \vdash B : \kappa' \rightsquigarrow^n_{A^\div} \mathbf{B}}{\Gamma \vdash \Pi\, \alpha : \kappa.\, B : \kappa' \rightsquigarrow^n_A \mathbf{\Pi}\, \boldsymbol{\alpha} : \boldsymbol{\kappa}.\, \mathbf{B}}\ [\mathrm{CPS}^n_A\text{-ProdK}]$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x = e : A \vdash B : \kappa' \rightsquigarrow^n_A \mathbf{B}}{\Gamma \vdash \mathsf{let}\, x = e : A\, \mathsf{in}\, B : \kappa' \rightsquigarrow^n_A \mathsf{let}\, \mathbf{x} = \mathbf{e} : \mathbf{A}\, \mathsf{in}\, \mathbf{B}}\ [\mathrm{CPS}^n_A\text{-Let}]$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : * \rightsquigarrow^n_{A^\div} \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : * \rightsquigarrow^n_A \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ [\mathrm{CPS}^n_A\text{-Sigma}]$$

$$\frac{\Gamma \vdash A : \kappa' \qquad \Gamma \vdash \kappa \equiv \kappa' \qquad \Gamma \vdash A : \kappa' \rightsquigarrow^n_A \mathbf{A}}{\Gamma \vdash A : \kappa \rightsquigarrow^n_A \mathbf{A}}\ [\mathrm{CPS}^n_A\text{-Conv}]$$

$\boxed{\Gamma \vdash A : * \rightsquigarrow^n_{A^\div} \mathbf{A}}$ Lemma 5.5 will show $\Gamma^+ \vdash A^\div : *^+$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^n_A \mathbf{A}}{\Gamma \vdash A : * \rightsquigarrow^n_{A^\div} \mathbf{\Pi}\, \boldsymbol{\alpha} : *.\, (\mathbf{A} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}}\ [\mathrm{CPS}^n_A\text{-Comp}]$$

Fig. 8. CPS^n of Universes, Kinds, and Types (excerpts)

as an η-expansion of a CPS'd computation. We must η-expand the variable case to guarantee CBN evaluation order, as we discuss shortly. In $[\mathrm{CPS}^n_e\text{-App}]$ we encode the CBN evaluation order for function application $e\ e'$ in the usual way. We translate the computations $e \rightsquigarrow^n_e \mathbf{e}$ and $e' \rightsquigarrow^n_e \mathbf{e'}$. First we evaluate $\mathbf{e}$ to a value $\mathbf{f}$, then apply $\mathbf{f}$ to the *computation* $\mathbf{e'}$. The application $\mathbf{f}\ \mathbf{e'}$ is itself a computation, which we call with the continuation $\mathbf{k}$.

Notice that only the translation rules $[\mathrm{CPS}^n_e\text{-Fst}]$ and $[\mathrm{CPS}^n_e\text{-Snd}]$ use the new @ form. As discussed in Section 3, to type check the translation of $\mathsf{snd}\ e$ produced by $[\mathrm{CPS}^n_e\text{-Snd}]$, we require the rule [T-Cont] when type checking the continuation that performs the second projection. While type checking the continuation, we know that the value $\mathbf{y}$ that the continuation receives is equivalent to

$\boxed{\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{A}}$ Lemma 5.5 will show $\Gamma^+ \vdash e^\div : A^\div$

$$\ldots \qquad \frac{\Gamma \vdash A : \kappa \rightsquigarrow^n_A \mathbf{A}}{\Gamma \vdash x : A \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{A} \to \boldsymbol{\alpha}.\, \mathbf{x}\ \boldsymbol{\alpha}\ \mathbf{k}}\ [\mathrm{CPS}^n_e\text{-Var}]$$

$$\frac{\Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa' \rightsquigarrow^n_{A^\div} \mathbf{B} \qquad \Gamma, x : A \vdash e : B \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : (\mathbf{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}) \to \boldsymbol{\alpha}.\, \mathbf{k}\ (\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{e})}\ [\mathrm{CPS}^n_e\text{-Fun}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \Pi\, x : A.\, B \rightsquigarrow^n_e \mathbf{e} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A}^\div \\ \Gamma, x : A \vdash B : \kappa' \rightsquigarrow^n_{A^\div} \mathbf{B}^\div \qquad \Gamma, x : A \vdash B : \kappa' \rightsquigarrow^n_A \mathbf{B}^+ \qquad \Gamma \vdash e' : A \rightsquigarrow^n_e \mathbf{e}'\end{array}}{\begin{array}{l}\Gamma \vdash e\ e' : B[e'/x] \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : (\mathbf{B}^+[\mathbf{e}'/\mathbf{x}]) \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{f} : \mathbf{\Pi}\, \mathbf{x} : \mathbf{A}^\div.\, \mathbf{B}^\div.\, (\mathbf{f}\ \mathbf{e}')\ \boldsymbol{\alpha}\ \mathbf{k})\end{array}}\ [\mathrm{CPS}^n_e\text{-App}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e} \\ \Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x = e : A \vdash B : \kappa' \rightsquigarrow^n_A \mathbf{B} \qquad \Gamma, x = e : A \vdash e' : B \rightsquigarrow^n_e \mathbf{e}'\end{array}}{\Gamma \vdash \mathsf{let}\, x = e : A\, \mathsf{in}\, e' : B[e/x] \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{B}[\mathbf{e}/\mathbf{x}] \to \boldsymbol{\alpha}.\, \mathsf{let}\, \mathbf{x} = \mathbf{e} : \mathbf{A}\, \mathsf{in}\, \mathbf{e}'\ \boldsymbol{\alpha}\ \mathbf{k}}\ [\mathrm{CPS}^n_e\text{-Let}]$$

$$\frac{\Gamma \vdash e_1 : A \rightsquigarrow^n_e \mathbf{e}_1 \qquad \Gamma \vdash e_2 : B[e_1/x] \rightsquigarrow^n_e \mathbf{e}_2 \qquad \Gamma \vdash A : * \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : * \rightsquigarrow^n_{A^\div} \mathbf{B}}{\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma\, x : A.\, B \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \to \boldsymbol{\alpha}.\, \mathbf{k}\ \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\ \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ [\mathrm{CPS}^n_e\text{-Pair}]$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \rightsquigarrow^n_{A^\div} \mathbf{A}^\div \qquad \Gamma, x : A \vdash B : * \rightsquigarrow^n_{A^\div} \mathbf{B}^\div \qquad \Gamma \vdash A : * \rightsquigarrow^n_A \mathbf{A}^+ \qquad \Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{fst}\, e : A \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{A}^+ \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}^\div.\, \mathbf{B}^\div.\, \mathsf{let}\, \mathbf{z} = \mathsf{fst}\, \mathbf{y}\, \mathsf{in}\, \mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k})\end{array}}\ [\mathrm{CPS}^n_e\text{-Fst}]$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \rightsquigarrow^n_{A^\div} \mathbf{A}^\div \qquad \Gamma, x : A \vdash B : * \rightsquigarrow^n_{A^\div} \mathbf{B}^\div \\ \Gamma, x : A \vdash B : * \rightsquigarrow^n_A \mathbf{B}^+ \qquad \Gamma \vdash (\mathsf{fst}\, e) : A \rightsquigarrow^n_e (\mathsf{fst}\, e)^\div \qquad \Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{snd}\, e : B[(\mathsf{fst}\, e)/x] \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{B}^+[(\mathsf{fst}\, e)^\div/\mathbf{x}] \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}^\div.\, \mathbf{B}^\div.\, \mathsf{let}\, \mathbf{z} = \mathsf{snd}\, \mathbf{y}\, \mathsf{in}\, \mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k})\end{array}}\ [\mathrm{CPS}^n_e\text{-Snd}]$$

$$\frac{\Gamma \vdash e : B \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e}}\ [\mathrm{CPS}^n_e\text{-Conv}]$$

$\boxed{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma}}$ Lemma 5.5 will show $\vdash \Gamma^+$

$$\frac{}{\vdash \cdot \rightsquigarrow^n \cdot}\ [\mathrm{CPS}^n_\Gamma\text{-Empty}] \qquad \frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A}}{\vdash \Gamma, x : A \rightsquigarrow^n \mathbf{\Gamma}, \mathbf{x} : \mathbf{A}}\ [\mathrm{CPS}^n_\Gamma\text{-AssumT}]$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash \kappa : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha : \kappa \rightsquigarrow^n \mathbf{\Gamma}, \boldsymbol{\alpha} : \boldsymbol{\kappa}}\ [\mathrm{CPS}^n_\Gamma\text{-AssumK}]$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^n_{A^\div} \mathbf{A} \qquad \Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e}}{\vdash \Gamma, x = e : A \rightsquigarrow^n \mathbf{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}}\ [\mathrm{CPS}^n_\Gamma\text{-Def}]$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash \kappa : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha = A : \kappa \rightsquigarrow^n \mathbf{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}}\ [\mathrm{CPS}^n_\Gamma\text{-DefT}]$$

Fig. 9. CPS^n of Terms and Environments (excerpts)

$e^{\div}\ \alpha\ \mathrm{id}$. Now, the reason we must use the @ syntax in the in the translation [CPS^n_e-Fst] is so that we can apply the [≡-Cont] rule to resolve the equivalence of the two *first projections* in the type of the second projection. That is, as we saw in Section 3, type preservation fails because we must show equivalence between $(\mathrm{fst}\ e)^{\div}$ and $\mathrm{fst}\ \mathrm{y}$. Since these are the only two cases that require our new rules, these are the only cases where we use the @ form in our translation; all other translation rules use standard application. In Section 6, we will see that the CBV translation must use the @ form much more frequently since, intuitively, our new equivalence rule recovers a notion of "value" in our CPS'd language, and in call-by-*value* types can only depend on values.

Our CPS translation encodes the CBN evaluation order explicitly so that the evaluation order of compiled terms is independent of the target language's evaluation order. This property is not immediately obvious since the [CPS^n_e-Let] rule binds a variable x to an expression e, making it seem like there are two possible evaluation orders: either evaluate e first, or substitute e for x first. Note, however, that our CBN translation always produces a λ term—even in the variable case since [CPS^n_e-Var] employs η-expansion as noted above. Therefore, in the [CPS^n_e-Let] rule e will always be a value, which means it doesn't evaluate in either CBN or CBV. Therefore, there is no ambiguity in how to evaluate the translation of let.

The translation rule [CPS^n_e-Conv] is deceptively simple. We could equivalently write this translation as follows, which makes its subtlety apparent.

$$\frac{\Gamma \vdash e : B \qquad \Gamma \vdash A \equiv B \qquad \Gamma \vdash e : B \rightsquigarrow^n_e \mathrm{e} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^n_A \mathrm{A} \qquad \Gamma \vdash B : \kappa \rightsquigarrow^n_A \mathrm{B}}{\Gamma \vdash e : A \rightsquigarrow^n_e \lambda\, \alpha : *.\, \lambda\, \mathrm{k} : \mathrm{A} \to \alpha.\, \mathrm{e}\ \alpha\ (\lambda\, \mathrm{x} : \mathrm{B}.\, \mathrm{k}\ \mathrm{x})}\ [\mathrm{CPS}^n_e\text{-Conv}]$$

Notice now that while the continuation k expects a term of type A, we call k with a term of type B. Intuitively, this should be fine since A and B should be equivalent, but formally this introduces a subtlety in the staging of our proof of type preservation, which we discuss next in Section 5.1.

We lift the translations to environments in the usual way, at the bottom of Figure 9. Since this is the CBN translation, we recur over the environment applying the *computation* translation.

5.1 Proof of Type Preservation for CPS^n

In a dependent type system, type preservation requires *coherence*, which essentially tells us that the translation preserves definitional equivalence. Since equivalence is defined by reduction, we first have to show that reduction sequences are preserved. Since reduction relies on substitution, we first must show *compositionality*, *i.e.*, that the translation commutes with substitution.

However, there is a problem with the proof architecture for CPS. Typed CPS for a *domain-full* target language inserts the type of every term into the output as a type annotation on the continuation. For example, $e : A$ is compiled to $\lambda\, \alpha : *.\, \lambda\, \mathrm{k} : A^+ \to \alpha.\, (\ldots)$. Therefore, the translation is defined on typing derivations, not on syntax. This introduces a problem in the case of the translation of the typing rule [Conv]. As alluded to above when describing [CPS^n_e-Conv], in order to preserve typing, we must first show coherence, *i.e.*, that we preserve equivalence. Working with the second definition we gave for the [CPS^n_e-Conv] rule, we need to show that the following term is well-typed.

$$\lambda\, \mathrm{k} : A^+ \to \alpha.\, e^{\div}\ \alpha\ (\lambda\, \mathrm{x} : B^+.\, \mathrm{k}\ \mathrm{x})$$

Note that this term seems to only make sense when $A^+ \equiv B^+$. While we have $A \equiv B$ from the source typing derivation, we don't know that $A^+ \equiv B^+$ unless we have coherence. But if equivalence is only defined on well-typed terms, as is the case in some dependently typed languages, we must first prove type preservation to know that A^+ and B^+ are well typed before we can prove coherence. So we have a circularity: type preservation requires coherence, but coherence requires type preservation.

A similar problem arises in other dependent typing rules, like the translation of application. In the case of application, to show type preservation we must show compositionality, *i.e.*, that

the translation commutes with substitution up to equivalence. Again we have a circularity: we need type preservation to prove compositionality, but to prove compositionality we need type preservation.

Barthe et al. [1999] explain this in detail, and their solution is to use a domain-free target language. This avoids the circularity because when there are no type annotations to generate, the translation can be defined on the syntax instead of typing derivations.

We solve the problem by using an untyped equivalence, which is based on the equivalence for the Calculus of Inductive Constructions from the Coq reference manual [The Coq Development Team 2017, Chapter 4]. Since the equivalence is untyped, we can show equivalence between terms with different domain annotations as long as their behavior is equivalent. This allows us to stage the proof: we can prove compositionality before coherence, and prove coherence before type preservation. While it seems surprising that we can prove equivalence between possibly ill-typed terms, recall that in the type system, we only appeal to the equivalence *after* checking that the terms we wish to prove equivalent are well typed. We can think of this equivalence as proving a stronger equivalence than we provide for well-typed terms, allowing us to prove a stronger form of coherence: in addition to preserving all well-typed equivalences, we also preserve certain equivalences that are valid according to their dynamic behavior, but conservatively ruled out by the strong type system. From this version of coherence, we are able to prove type preservation.

The proofs in this section are staged as follows. First we show compositionality (Lemma 5.1), since reduction is defined in terms of substitution. Then we show that the translation preserves reduction sequences (Lemma 5.2 and Lemma 5.3), which allows us to show coherence (Lemma 5.4). Using compositionality and coherence, we prove that the translation is type preserving (Lemma 5.5).

We now show Lemma 5.1, which states that the CPS^n translation commutes with substitution. The formal statement of the lemma is somewhat complicated since we have the cross product of four syntactic categories and two translations. However, the intuition is simple: first substituting and then translating is equivalent to translating and then substituting.

This lemma is critical to our proofs. Since reduction is essentially defined by substitution, this lemma does most of the work in showing that the translation preserves reduction. However, it is also necessary in type preservation when showing that a dependent type is preserved. A dependent type, such as $B[e'/x]$ produced by the rule [App], occurs when we perform substitution into a type. We want to show, for example, that if $e : B[e'/x]$ then the translation of e has the type translation $(B[e'/x])^\div$. Since our translation is compositional, *i.e.*, commutes with substitution, we know how to translate $B[e'/x]$ by simply translating B and e'.

Lemma 5.1 (CPS^n Compositionality).

(1) $(\kappa[A/\alpha])^+ \equiv \kappa^+[A^+/\boldsymbol{\alpha}]$
(2) $(\kappa[e/x])^+ \equiv \kappa^+[e^\div/\mathbf{x}]$
(3) $(A[B/\alpha])^+ \equiv A^+[B^+/\boldsymbol{\alpha}]$
(4) $(A[e/x])^+ \equiv A^+[e^\div/\mathbf{x}]$
(5) $(A[B/\alpha])^\div \equiv A^\div[B^+/\boldsymbol{\alpha}]$
(6) $(A[e/x])^\div \equiv A^\div[e^\div/\mathbf{x}]$
(7) $(e[A/\alpha])^\div \equiv e^\div[A^+/\boldsymbol{\alpha}]$
(8) $(e[e'/x])^\div \equiv e^\div[e'^\div/\mathbf{x}]$

Proof. In the PTS syntax, we represent source expressions as $t[t'/x]$. The proof is by induction on the typing derivations for t. Note that our $^\div$ and $^+$ notation implicitly require the typing derivations as premises. The proof is completely straightforward.

Case [Conv]. The proof is trivial, now that we have staged the proof appropriately. We give part 8 as an example.

$$\frac{\Gamma \vdash e : B \qquad \Gamma \vdash A : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : A}$$

We must show that $(e[e'/x])^\div \equiv e^\div[e'^\div/\mathbf{x}]$ (at type A). Note that by part 8 of the induction hypothesis, we know that $(e[e'/x])^\div \equiv e^\div[e'^\div/\mathbf{x}]$ (at the smaller derivation for type B). But recall that our equivalence cares nothing for types, so the proof is complete. □

We next prove that the translation preserves reduction sequences, Lemma 5.2 and Lemma 5.3. Note that kinds do not take steps in the one step reduction, but can in the $\rhd^\star$ relation since it reduces under all contexts. As mentioned earlier, this is necessary to show equivalence is preserved, since equivalence is defined in terms of reduction.

Lemma 5.2 (CPS^n Preserves One-Step Reduction).

- *If* $\Gamma \vdash e : A$ *and* $e \rhd e'$ *then* $e^\div \rhd^\star \mathbf{e'}$ *and* $\mathbf{e'} \equiv e'^\div$
- *If* $\Gamma \vdash A : \kappa$ *and* $A \rhd A'$ *then* $A^+ \rhd^\star \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^+$
- *If* $\Gamma \vdash A : *$ *and* $A \rhd A'$ *then* $A^\div \rhd^\star \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^\div$

Proof. The proof is straightforward by cases on the $\rhd$ relation. □

Lemma 5.3 (CPS^n Preserves Reduction Sequences).

- *If* $\Gamma \vdash e : A$ *and* $e \rhd^\star e'$ *then* $e^\div \rhd^\star \mathbf{e'}$ *and* $\mathbf{e'} \equiv e'^\div$
- *If* $\Gamma \vdash A : \kappa$ *and* $A \rhd^\star A'$ *then* $A^+ \rhd^\star \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^+$
- *If* $\Gamma \vdash A : *$ *and* $A \rhd^\star A'$ *then* $A^\div \rhd^\star \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^\div$
- *If* $\Gamma \vdash \kappa : U$ *and* $\kappa \rhd^\star \kappa'$ *then* $\kappa^+ \rhd^\star \boldsymbol{\kappa'}$ *and* $\boldsymbol{\kappa'} \equiv \kappa'^+$

Proof. The proof is straightforward by induction on the length of the reduction sequence. The base case is trivial and the inductive case follows by Lemma 5.2 and the inductive hypothesis. □

Lemma 5.4 (CPS^n Coherence).

- *If* $e \equiv e'$ *then* $e^\div \equiv e'^\div$
- *If* $A \equiv A'$ *then* $A^+ \equiv A'^+$
- *If* $A \equiv A'$ *then* $A^\div \equiv A'^\div$
- *If* $\kappa \equiv \kappa'$ *then* $\kappa^+ \equiv \kappa'^+$

Proof. The proof is by induction on the derivation of $e \equiv e'$. The base case follows by Lemma 5.3, and the cases of η-equivalence follow from Lemma 5.3, the induction hypothesis, and the fact that we have the same η-equivalence rules in the CC^k. □

We first prove type preservation, Lemma 5.5, using the explicit syntax on which we defined CPS^n. This proof is our central contribution to the call-by-name translation. In this lemma, proving that the translation of snd e preserves typing requires both the new typing rule [T-Cont] and the equivalence rule [≡-Cont]. The rest of the proof is straightforward. For full details, see our online supplementary materials [Bowman et al. 2017].

Lemma 5.5 (CPS^n is Type Preserving (Explicit Syntax)).

(1) *If* $\vdash \Gamma$ *then* $\vdash \Gamma^+$
(2) *If* $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash e^\div : A^\div$
(3) *If* $\Gamma \vdash A : \kappa$ *then* $\Gamma^+ \vdash A^+ : \kappa^+$
(4) *If* $\Gamma \vdash A : *$ *then* $\Gamma^+ \vdash A^\div : *^+$
(5) *If* $\Gamma \vdash \kappa : U$ *then* $\Gamma^+ \vdash \kappa^+ : U^+$

Proof. All cases are proven simultaneously by mutual induction on the type derivation and well-formedness derivation. Part 4 follows easily by part 3 in every case, so we elide its proof. Most cases follow easily from the induction hypotheses.

Case [Snd] $\Gamma \vdash \mathsf{snd}\ e : B[\mathsf{fst}\ e/x]$

We must show that $\lambda\ \alpha : *.\ \lambda\ \mathsf{k} : B^+[(\mathsf{fst}\ e)^\div/\mathsf{x}] \to \alpha.$
$$e^\div\ @\ \alpha\ (\lambda\ \mathsf{y} : \Sigma\ \mathsf{x} : A^\div.\ B^\div.\ \mathsf{let\ z = snd\ y\ in\ z}\ \alpha\ \mathsf{k})$$
has type $(B[\mathsf{fst}\ e/x])^\div$.

By part 6 of Lemma 5.1, and definition of the translation, this type is equivalent to
$\Pi\ \alpha : *.\ (B^+[(\mathsf{fst}\ e)^\div/\mathsf{x}] \to \alpha) \to \alpha$

By [Lam], it suffices to show that

$$\Gamma^+, \alpha : *, \mathsf{k} : B^+[(\mathsf{fst}\ e)^\div/x] \to \alpha \vdash e^\div\ @\ \alpha\ (\lambda\ \mathsf{y} : \Sigma\ \mathsf{x} : A^\div.\ B^\div.\ \mathsf{let\ z = snd\ y\ in\ z}\ \alpha\ \mathsf{k}) : \alpha$$

This is the key difficulty in the proof. Recall from Section 3 that the term $\mathsf{z}\ \alpha$ has type $(B^+[\mathsf{fst\ y}/\mathsf{x}] \to \alpha) \to \alpha$ while the term k has type $B^+[(\mathsf{fst}\ e)^\div/\mathsf{x}] \to \alpha$. To show that $\mathsf{z}\ \alpha\ \mathsf{k}$ is well-typed, we must show that $(\mathsf{fst}\ e)^\div \equiv \mathsf{fst\ y}$. We proceed by our new typing rule [T-Cont], which will help us prove this.

First, note that $e^\div\ (\Sigma\ \mathsf{x} : A^\div.\ B^\div)\ \mathsf{id}$ is well-typed. By part 4 of the induction hypothesis we know that $\Gamma^+ \vdash A^\div : *$ and $\Gamma^+, \mathsf{x} : A^\div \vdash B^\div : *$. By part 2 of the induction hypothesis applied to $\Gamma \vdash e : \Sigma\ x : A.\ B$, we know $\Gamma^+ \vdash e^\div : \Pi\ \alpha : *.\ (\Sigma\ \mathsf{x} : A^\div.\ B^\div \to \alpha) \to \alpha$.

Now, by [T-Cont], it suffices to show that

$$\Gamma^+, \alpha : *, \mathsf{k} : B^+[(\mathsf{fst}\ e)^\div/\mathsf{x}] \to \alpha, \mathsf{y} = e^\div\ \Sigma\ \mathsf{x} : A^\div.\ B^\div\ \mathsf{id} \vdash \mathsf{let\ z = snd\ y\ in\ z}\ \alpha\ \mathsf{k} : \alpha$$

Note that we now have the definitional equivalence $\mathsf{y} = e^\div\ (\Sigma\ \mathsf{x} : A^\div.\ B^\div)\ \mathsf{id}$. By [Let] it suffices to show

$$\Gamma^+, \alpha : *, \mathsf{k} : B^+[(\mathsf{fst}\ e)^\div/\mathsf{x}] \to \alpha, \mathsf{y} = e^\div\ \Sigma\ \mathsf{x} : A^\div.\ B^\div\ \mathsf{id}, \mathsf{z} = \mathsf{snd\ y} : B^\div[\mathsf{fst\ y}/\mathsf{x}] \vdash \mathsf{z}\ \alpha\ \mathsf{k} : \alpha$$

Note that

$$\begin{aligned}
&\mathsf{z} : B^\div[\mathsf{fst\ y}/\mathsf{x}] && && (11)\\
&= \Pi\ \alpha : *.\ (B^+[\mathsf{fst\ y}/\mathsf{x}] \to \alpha) \to \alpha && \text{by definition of } B^\div && (12)\\
&\equiv \Pi\ \alpha : *.\ (B^+[\mathsf{fst}\ (e^\div\ _\ \mathsf{id})/\mathsf{x}] \to \alpha) \to \alpha && \text{by } \delta \text{ reduction on } \mathsf{y} && (13)
\end{aligned}$$

The Equation (13) above, in which we δ reduce y, is impossible without [T-Cont].

By [Conv], and since $\mathsf{k} : B^+[(\mathsf{fst}\ e)^\div/\mathsf{x}] \to \alpha$, to show $\mathsf{z}\ \alpha\ \mathsf{k} : \alpha$ it suffices to show that $(\mathsf{fst}\ e)^\div \equiv \mathsf{fst}\ (e^\div\ \_\ \mathsf{id})$.

Note that $(\mathsf{fst}\ e)^\div = \lambda\ \alpha : *.\ \lambda\ \mathsf{k}' : (A^+ \to \alpha).$
$$e^\div\ @\ \alpha\ (\lambda\ \mathsf{y} : \Sigma\ \mathsf{x} : A^\div.\ B^\div.\ \mathsf{let\ z' = fst\ y} : A^\div\ \mathsf{in\ z'}\ \alpha\ \mathsf{k}')$$
by definition of the translation.

By [$\equiv$-η], it suffices to show that

$$\begin{aligned}
&e^\div\ @\ \alpha\ (\lambda\ \mathsf{y} : \Sigma\ \mathsf{x} : A^\div.\ B^\div.\ \mathsf{let\ z' = fst\ y} : A^\div\ \mathsf{in\ z'}\ \alpha\ \mathsf{k}') && && (14)\\
&\equiv (\lambda\ \mathsf{y} : \Sigma\ \mathsf{x} : A^\div.\ B^\div.\ \mathsf{let\ z' = fst\ y\ in\ z'}\ \alpha\ \mathsf{k}')\ (e^\div\ _\ \mathsf{id}) && [\equiv\text{-Cont}] && (15)\\
&\equiv (\mathsf{fst}\ (e^\div\ _\ \mathsf{id}))\ \alpha\ \mathsf{k}' && \text{by reduction} && (16)
\end{aligned}$$

Notice that Equation (15) requires our new equivalence rule applied to the translation of the fst.

Case [Conv] $\Gamma \vdash e : A$ such that $\Gamma \vdash e : B$ and $A \equiv B$.

We must show that $e^\div$ has type $A^\div = \Pi\ \alpha : *.\ (A^+ \to \alpha) \to \alpha$.

By the induction hypothesis, we know that $e^\div : B^\div = \Pi\ \alpha : *.\ (B^+ \to \alpha) \to \alpha$. By [Conv] it suffices to show that $A^+ \equiv B^+$, which follows by Lemma 5.4. □

To recover a simple statement of the type-preservation theorem over the PTS syntax, we define two meta-functions for translating terms and types depending on their use. We define cps $[\![t]\!]$ to translate a PTS expression in "term" position, *i.e.*, when used on the left side of a type annotation as in $t : t'$, and we define cpsT $[\![t']\!]$ to translate an expression in "type" position, *i.e.*, when used on

the right side of a type annotation. We define these in terms of the translation shown above, noting that for every $t : t'$ in the PTS syntax, one of the following is true: t is a term e and t' is a type A in the explicit syntax; t is a type A and t' is a kind κ in the explicit syntax; or t is a kind κ and t' is a universe U in the explicit syntax.

$$\begin{array}{llll} \textsc{cps}\,[\![t]\!] \stackrel{\text{def}}{=} e^\div \text{ when } t \text{ is a term} & & \textsc{cpsT}\,[\![t']\!] \stackrel{\text{def}}{=} A^\div \text{ when } t' \text{ is a type} \\ \textsc{cps}\,[\![t]\!] \stackrel{\text{def}}{=} A^+ \text{ when } t \text{ is a type} & & \textsc{cpsT}\,[\![t']\!] \stackrel{\text{def}}{=} \kappa^+ \text{ when } t' \text{ is a kind} \\ \textsc{cps}\,[\![t]\!] \stackrel{\text{def}}{=} \kappa^+ \text{ when } t \text{ is a kind} & & \textsc{cpsT}\,[\![t']\!] \stackrel{\text{def}}{=} U^+ \text{ when } t' \text{ is a universe} \end{array}$$

This notation is based on Barthe and Uustalu [2002].

Theorem 5.6 (CPS^n is Type Preserving (PTS syntax)). *$\Gamma \vdash t : t'$ then $\Gamma^+ \vdash \textsc{cps}\,[\![t]\!] : \textsc{cpsT}\,[\![t']\!]$*

5.2 Proof of Correctness for CPS^n

Since type preservation in a dependently typed language requires preserving reduction sequences, we have already done most of the work to prove two other compiler correctness properties: correctness of separate compilation, and whole-program compiler correctness. To specify compiler correctness, we need an independent specification that tells us how source values—or, more generally, observations—are related to target values. For instance, when compiling to C we might specify that the number 5 is related to the bits $0x101$. Without a specification, independent of the compiler, there is no definition that the compiler can be correct with respect to. In our setting, such an independent specification is simple to construct. We can add ground values such as booleans to our language with the obvious cross-language relation: $\mathsf{True} \approx \mathbf{True}$ and $\mathsf{False} \approx \mathbf{False}$.

Next, to define separate compilation, we need a definition of linking. We can define linking as substitution, and define valid closing substitutions γ as follows.

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

We extend the compiler in a straightforward way to compile closing substitutions, written $\gamma^\div$, and allow compiled code to be linked with the compilation of any valid closing substitution γ. This definition supports a separate compilation theorem that allows linking with the output of our compiler, but not with the output of other compilers.

Now we can show that the compiler is correct with respect to separate compilation: if we first link and run to a value, we get a related value when we compile and then link with the compiled closing substitution. Since our target language is in CPS form, we should apply the halt continuation, $\mathbf{id}$, and compare the ground values.

Theorem 5.7 (Separate Compilation Correctness). *If $\Gamma \vdash e : A$ where A is ground, and $\gamma(e) \rhd^\star v$ then $\gamma^\div(e^\div)\ A^+\ \mathbf{id} \rhd^\star \mathbf{v}$ and $v \approx \mathbf{v}$.*

Proof. Since reduction implies equivalence, we reason in terms of equivalence. By Lemma 5.3, $(\gamma(e))^\div \rhd^\star \mathbf{e}$ and $v^\div \equiv \mathbf{e}$. By Lemma 5.1, $(\gamma(e))^\div \equiv \gamma^\div(e^\div)$, hence $\gamma^\div(e^\div) \rhd^\star \mathbf{e}$ and $v^\div \equiv \mathbf{e}$. Since the translation on all ground values is $v^\div = \boldsymbol{\lambda}\,\alpha.\,\boldsymbol{\lambda}\,\mathbf{k}.\,\mathbf{k}\ \mathbf{v}$, where $v \approx \mathbf{v}$, we know $v^\div\ A^+\ \mathbf{id} \rhd^\star \mathbf{v}$ such that $v \approx \mathbf{v}$. Since $v^\div \equiv \mathbf{e} \equiv \gamma^\div(e^\div)$, we also know that $\gamma^\div(e^\div)\ A^+\mathbf{id} \rhd^\star \mathbf{v}'$ and $\mathbf{v}' \equiv \mathbf{v}$. Since $\mathbf{v}$ is ground, $\mathbf{v}' = \mathbf{v}$ and $v \approx \mathbf{v}'$. □

Corollary 5.8 (Whole-Program Compiler Correctness). *If $\vdash e : A$ and $e \rhd^\star v$ then $e^\div\ A^+\ \mathbf{id} \rhd^\star \mathbf{v}$ and $v \approx \mathbf{v}$.*

Our separate-compilation correctness theorem is similar to the guarantees provided by SepCompCert [Kang et al. 2016] in that it supports linking with only the output of the same compiler. We could support more flexible notions of linking—such as linking with code produced by different compilers, from different source languages, or code written directly in the target language—by

$\boxed{\Gamma \vdash \kappa : U \rightsquigarrow^v_\kappa \boldsymbol{\kappa}}$ Lemma 6.3 will show $\Gamma^+ \vdash \kappa^+ : U^+$

$$\ldots \qquad \frac{\Gamma \vdash A : \kappa' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash \kappa : U \rightsquigarrow^v_\kappa \boldsymbol{\kappa}}{\Gamma \vdash \Pi\, x : A.\, \kappa : U \rightsquigarrow^v_\kappa \mathbf{\Pi\, x : A.\, \boldsymbol{\kappa}}}\ [\mathrm{CPS}^v_\kappa\text{-ProdA}]$$

$\boxed{\Gamma \vdash A : \kappa \rightsquigarrow^v_A \mathbf{A}}$ Lemma 6.3 will show $\Gamma^+ \vdash A^+ : \kappa^+$

$$\ldots \qquad \frac{\Gamma \vdash A : \kappa' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \lambda\, x : A.\, B : \Pi\, x : A.\, \kappa \rightsquigarrow^v_A \mathbf{\lambda\, x : A.\, B}}\ [\mathrm{CPS}^v_A\text{-Constr}]$$

$$\frac{\Gamma \vdash A : \Pi\, x : B.\, \kappa \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa' \rightsquigarrow^v_A \mathbf{B} \qquad \Gamma \vdash e : B \rightsquigarrow^v_e \mathbf{e}}{\Gamma \vdash A\, e : \kappa[e/x] \rightsquigarrow^v_A \mathbf{A\ (e\ B\ id)}}\ [\mathrm{CPS}^v_A\text{-AppConstr}]$$

$$\frac{\Gamma \vdash A : \kappa' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa \rightsquigarrow^v_{A^\div} \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \kappa \rightsquigarrow^v_A \mathbf{\Pi\, x : A.\, B}}\ [\mathrm{CPS}^v_A\text{-Prod}]$$

$$\frac{\Gamma \vdash \kappa : U' \rightsquigarrow^v_\kappa \boldsymbol{\kappa} \qquad \Gamma, x : A \vdash B : U \rightsquigarrow^v_{A^\div} \mathbf{B}}{\Gamma \vdash \Pi\, \alpha : \kappa.\, B : U \rightsquigarrow^v_A \mathbf{\Pi\, \alpha : \kappa.\, B}}\ [\mathrm{CPS}^v_A\text{-ProdK}]$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow^v_e \mathbf{e} \qquad \Gamma \vdash A : \kappa' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x = e : A \vdash B : \kappa \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \mathsf{let}\, x = e : A\, \mathsf{in}\, B : \kappa \rightsquigarrow^v_A \mathbf{let\ x = e\ A\ id : A\ in\ B}}\ [\mathrm{CPS}^v_A\text{-Let}]$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : * \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : * \rightsquigarrow^v_A \mathbf{\Sigma\, x : A.\, B}}\ [\mathrm{CPS}^v_A\text{-Sigma}]$$

Fig. 10. CPS^v of Kinds and Types (excerpts)

defining an independent specification for when closing substitutions are related across languages (*e.g.*, [Ahmed and Blume 2011; Neis et al. 2015; New et al. 2016; Perconti and Ahmed 2014]).

6 CALL-BY-VALUE CPS TRANSLATION OF CC

In this section, we present the call-by-value CPS translation (CPS^v) of CC. First, we redefine our short-hand from Section 5 to refer to call-by-value translation.

$$A^\div \stackrel{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash A : * \rightsquigarrow^v_{A^\div} \mathbf{A} \qquad U^+ \stackrel{\text{def}}{=} \mathbf{U} \text{ where } \Gamma \vdash U \rightsquigarrow^v_U \mathbf{U}$$
$$e^\div \stackrel{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash e : A \rightsquigarrow^v_e \mathbf{e} \qquad \kappa^+ \stackrel{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash \kappa : U \rightsquigarrow^v_\kappa \boldsymbol{\kappa}$$
$$A^+ \stackrel{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash A : \kappa \rightsquigarrow^v_A \mathbf{A}$$

Unlike CBN, the CBV translation forces every computation to a value. Therefore, every dependent elimination requires our new [T-Cont] typing rule. Moreover, all substitutions of a term into a type must substitute *values* instead of computations so all dependent type annotations must explicitly convert computations to values by supplying the identity continuation.

We present our call-by-value translation CPS^v in Figures 10 and 11. In general, CPS^v differs from CPS^n in two ways. First, all term variables must have value types, so the translation rules for all binding constructs now use the value translation for type annotations. Second, we change the definition of value types so that functions must receive values and pairs must contain values.

The universe translation is unchanged from CPS^n, so we omit it here. The kind translation (Figure 10) has changed in only one place. Now the translation rule $[\mathrm{CPS}^v_\kappa\text{-ProdA}]$ translates the kind of type-level functions $\Pi\, x : A.\, \kappa$ to accept a *value* as argument $\mathbf{x} : A^+$.

The type translation (Figure 10) has multiple rules with variable annotations that have changed from CBN. The computation translation of types is unchanged. In the value translation, similar to

$\boxed{\Gamma \vdash e : A \rightsquigarrow^{v}_{e} \mathbf{e}}$ Lemma 6.3 will show $\Gamma^{+} \vdash e^{\div} : A^{\div}$

$$\ldots \qquad \frac{\Gamma \vdash A : \kappa \rightsquigarrow^{v}_{A} \mathbf{A}}{\Gamma \vdash x : A \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{A} \to \boldsymbol{\alpha}.\, \mathbf{k}\ \mathbf{x}}\ [\mathrm{CPS}^{v}_{e}\text{-Var}]$$

$$\frac{\Gamma \vdash A : \kappa' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, x : A \vdash B : \kappa \rightsquigarrow^{v}_{A^{\div}} \mathbf{B} \qquad \Gamma, x : A \vdash e : B \rightsquigarrow^{v}_{e} \mathbf{e}}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : (\mathbf{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}) \to \boldsymbol{\alpha}.\, \mathbf{k}\ (\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{e})}\ [\mathrm{CPS}^{v}_{e}\text{-Fun}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \Pi\, x : A.\, B \rightsquigarrow^{v}_{e} \mathbf{e} \\ \Gamma, x : A \vdash B : \kappa \rightsquigarrow^{v}_{A^{\div}} \mathbf{B}^{\div} \qquad \Gamma, x : A \vdash B : \kappa \rightsquigarrow^{v}_{A} \mathbf{B}^{+} \qquad \Gamma \vdash e' : A \rightsquigarrow^{v}_{e} \mathbf{e}' \qquad \Gamma \vdash A : \kappa' \rightsquigarrow^{v}_{A} \mathbf{A}\end{array}}{\begin{array}{l}\Gamma \vdash e\ e' : B[e'/x] \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : (\mathbf{B}^{+}[(\mathbf{e}'\ \mathbf{A}\ \mathrm{id})/\mathbf{x}]) \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{f} : \mathbf{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}^{\div}. \\ \qquad\quad \mathbf{e}'\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, (\mathbf{f}\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k}))\end{array}}\ [\mathrm{CPS}^{v}_{e}\text{-App}]$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow^{v}_{e} \mathbf{e} \qquad \Gamma \vdash A : \kappa' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma \vdash B : \kappa \rightsquigarrow^{v}_{A} \mathbf{B} \qquad \Gamma, x = e : A \vdash e' : B \rightsquigarrow^{v}_{e} \mathbf{e}'}{\begin{array}{l}\Gamma \vdash \mathrm{let}\ x = e : A\ \mathrm{in}\ e' : B[e/x] \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{B}[(\mathbf{e}\ \mathbf{A}\ \mathrm{id})/\mathbf{x}] \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{e}'\ \boldsymbol{\alpha}\ \mathbf{k})\end{array}}\ [\mathrm{CPS}^{v}_{e}\text{-Let}]$$

$$\frac{\Gamma \vdash e_1 : A \rightsquigarrow^{v}_{e} \mathbf{e}_1 \qquad \Gamma \vdash e_2 : B[e_1/x] \rightsquigarrow^{v}_{e} \mathbf{e}_2 \qquad \Gamma \vdash A : * \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, x : A \vdash B : * \rightsquigarrow^{v}_{A} \mathbf{B}}{\begin{array}{l}\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma\, x : A.\, B \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}_1\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{x}_1 : \mathbf{A}.\, \mathbf{e}_2\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{x}_2 : \mathbf{B}[(\mathbf{e}_1\ \mathbf{A}\ \mathrm{id})/\mathbf{x}]. \\ \qquad\quad \mathbf{k}\ \langle \mathbf{x}_1, \mathbf{x}_2 \rangle\ \mathrm{as}\ \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}))\end{array}}\ [\mathrm{CPS}^{v}_{e}\text{-Pair}]$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow^{v}_{e} \mathbf{e}}{\begin{array}{l}\Gamma \vdash \mathrm{fst}\ e : A \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{A}^{+} \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}.\, \mathrm{let}\ \mathbf{z} = \mathbf{fst}\ \mathbf{y}\ \mathrm{in}\ \mathbf{k}\ \mathbf{z})\end{array}}\ [\mathrm{CPS}^{v}_{e}\text{-Fst}]$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \rightsquigarrow^{v}_{A} \mathbf{A} \\ \Gamma, x : A \vdash B : * \rightsquigarrow^{v}_{A} \mathbf{B} \qquad \Gamma \vdash (\mathrm{fst}\ e) : A \rightsquigarrow^{v}_{e} (\mathrm{fst}\ e)^{\div} \qquad \Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow^{v}_{e} \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathrm{snd}\ e : B[\mathrm{fst}\ e/x] \rightsquigarrow^{v}_{e} \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{B}[((\mathbf{fst}\ e)^{\div}\ \mathbf{A}\ \mathrm{id})/\mathbf{x}] \to \boldsymbol{\alpha}. \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \mathbf{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}.\, \mathrm{let}\ \mathbf{z} = \mathbf{snd}\ \mathbf{y}\ \mathrm{in}\ \mathbf{k}\ \mathbf{z})\end{array}}\ [\mathrm{CPS}^{v}_{e}\text{-Snd}]$$

$$\frac{\Gamma \vdash e : B \rightsquigarrow^{v}_{e} \mathbf{e}}{\Gamma \vdash e : A \rightsquigarrow^{v}_{e} \mathbf{e}}\ [\mathrm{CPS}^{v}_{e}\text{-Conv}]$$

$\boxed{\vdash \Gamma \rightsquigarrow^{v} \mathbf{\Gamma}}$ Lemma 6.3 will show $\vdash \Gamma^{+}$

$$\frac{}{\vdash \cdot \rightsquigarrow^{v} \cdot}\ [\mathrm{CPS}^{v}_{\Gamma}\text{-Empty}] \qquad \frac{\vdash \Gamma \rightsquigarrow^{v} \mathbf{\Gamma} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^{v}_{A} \mathbf{A}}{\vdash \Gamma, x : A \rightsquigarrow^{v} \mathbf{\Gamma}, \mathbf{x} : \mathbf{A}}\ [\mathrm{CPS}^{v}_{\Gamma}\text{-AssumT}]$$

$$\frac{\vdash \Gamma \rightsquigarrow^{v} \mathbf{\Gamma} \qquad \Gamma \vdash \kappa : U \rightsquigarrow^{v}_{\kappa} \boldsymbol{\kappa}}{\vdash \Gamma, \alpha : \kappa \rightsquigarrow^{v} \mathbf{\Gamma}, \boldsymbol{\alpha} : \boldsymbol{\kappa}}\ [\mathrm{CPS}^{v}_{\Gamma}\text{-AssumK}]$$

$$\frac{\vdash \Gamma \rightsquigarrow^{v} \mathbf{\Gamma} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma \vdash e : A \rightsquigarrow^{v}_{e} \mathbf{e}}{\vdash \Gamma, x = e : A \rightsquigarrow^{v} \mathbf{\Gamma}, \mathbf{x} = \mathbf{e}\ \mathbf{A}\ \mathrm{id} : \mathbf{A}}\ [\mathrm{CPS}^{v}_{\Gamma}\text{-Def}]$$

$$\frac{\vdash \Gamma \rightsquigarrow^{v} \mathbf{\Gamma} \qquad \Gamma \vdash A : \kappa \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma \vdash \kappa : U \rightsquigarrow^{v}_{\kappa} \boldsymbol{\kappa}}{\vdash \Gamma, \alpha = A : \kappa \rightsquigarrow^{v} \mathbf{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}}\ [\mathrm{CPS}^{v}_{\Gamma}\text{-DefT}]$$

Fig. 11. CPS^{v} of Terms and Environments (excerpts)

the kind translation, dependent function types that abstract over terms now translate the argument annotation $x : A$ using the value translation. Dependent pair types $\Sigma\, x : A.\, B$ now translate to pairs of values $\Sigma\, \mathbf{x} : A^+.\, B^+$. When terms appear in the type language, such as in [CPS^v_A-AppConstr] and [CPS^v_A-Let], we must explicitly convert the computation to a value to maintain the invariant that all term variables refer to term values. Hence in [CPS^v_A-AppConstr] we translate a type-level application with a term argument $A\ e$ to $A^+\ (e^\div\ B^+\ \mathbf{id})$. We similarly translate let-bound terms [CPS^v_A-Let] by casting the computation to a value. Recall from Section 3 that while expressions of the form $A^+\ (e^\div\ B^+\ \mathbf{id})$ are not in CPS form, this expression is a type and will be evaluated during type checking. Terms that evaluate at run time are always in proper CPS form and do not return.

The term translation (Figure 11) changes in three major ways. As in Section 5, we implicitly have a computation and a value translation on term values, with the latter inlined into the former. First, unlike in CPS^n, variables are values, whereas the translation must produce a computation. Therefore, we translate x by "value η-expansion" into $\boldsymbol{\lambda}\, \boldsymbol{\alpha}.\, \boldsymbol{\lambda}\, \mathbf{k}.\, \mathbf{k}\ \mathbf{x}$, a computation that immediately applies the continuation to the value. Second, as discussed above, we change the translation of application [CPS^v_e-App] to force the evaluation of the function argument. Third and finally, in the translation of pairs [CPS^v_e-Pair], we force the evaluation of the components of the pair and produce a pair of values for the continuation. Note that in cases of the translation where we have types with dependency—[CPS^v_e-App], [CPS^v_e-Let], [CPS^v_e-Pair], and [CPS^v_e-Snd]—we cast computations to values in the types by applying the identity continuation, and require the @ form to use our new typing rule.

Given the translation of binding constructs in the language, the translation of the environment (Figure 11) should be unsurprising. Since all variables are values, we translate term variables $x : A$ using the value translation on types to produce $\mathbf{x} : A^+$ instead of $\mathbf{x} : A^\div$. We must also translate term definitions $x = e : A$ by casting the computation to a value, producing $\mathbf{x} = e^\div\ A^+\ \mathbf{id} : A^+$.

6.1 Proof of Type-Preservation for CPS^v

The structure of the type-preservation proof is the same as in Section 5. First we prove that the translation commutes with substitution, then that reduction sequences are preserved, then that equivalence is preserved, and finally that typing is preserved. Except for Lemma 6.2 (CPS^v Compositionality), the proofs of the supporting lemmas are essentially the same as in Section 5. We elide these lemmas and their proofs for brevity. For full details, see our online supplementary materials [Bowman et al. 2017].

We begin with a technical lemma that is essentially an η principle for CPS'd computations. In the CPS^v setting, we must frequently reason about normal forms of CPS'd computations. This lemma provides a useful abstraction for doing so.[5] The lemma states that any CPS'd computation $e^\div$ is equivalent to a new CPS'd computation that accepts a continuation $\mathbf{k}$ simply applies $e^\div$ to $\mathbf{k}$. The proof is straightforward. Note the type annotations are mismatched, as in our explanation of coherence in Section 5.1, but the behaviors of the terms are the same and equivalence is untyped.

Lemma 6.1 (CPS^v Computation η). $e^\div \equiv \boldsymbol{\lambda}\, \boldsymbol{\alpha} : *.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{A} \to \boldsymbol{\alpha}.\, e^\div\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{B}.\, \mathbf{k}\ \mathbf{x})$

Since variables are values in call-by-value, we adjust the statement of Lemma 6.2 to cast computations to values. Proving this lemma now requires our new equivalence rule [≡-Cont] for cases involving substitution of terms. By convention, all terms being translated have an implicit typing derivation, so the omitted types are easy to reconstruct.

[5] The proofs for the CBN setting only require a specialized instance of this property although the general form holds.

Lemma 6.2 (CPS^v Compositionality).

(1) $(\kappa[A/\alpha])^+ \equiv \kappa^+[A^+/\boldsymbol{\alpha}]$
(2) $(\kappa[e/x])^+ \equiv \kappa^+[e^\div \; \_ \; \mathbf{id}/\mathbf{x}]$
(3) $(A[B/\alpha])^+ \equiv A^+[B^+/\boldsymbol{\alpha}]$
(4) $(A[e/x])^+ \equiv A^+[e^\div \; \_ \; \mathbf{id}/\mathbf{x}]$
(5) $(A[B/\alpha])^\div \equiv A^\div[B^+/\boldsymbol{\alpha}]$
(6) $(A[e/x])^\div \equiv A^\div[e^\div \; \_ \; \mathbf{id}/\mathbf{x}]$
(7) $(e[A/\alpha])^\div \equiv e^\div[A^+/\boldsymbol{\alpha}]$
(8) $(e[e'/x])^\div \equiv e^\div[e'^\div \; \_ \; \mathbf{id}/\mathbf{x}]$

Proof. The proof is straightforward by induction on the typing derivation of the expression t being substituted into. We give one representative case.

Case [Var] Part 8, $(x[e/x'])^\div$
We must show $(x[e/x'])^\div = x^\div[e^\div \; \_ \; \mathbf{id}/\mathbf{x}']$.
W.l.o.g., assume $x = x'$.

$$(x[e/x])^\div \tag{17}$$
$$= e^\div \qquad \text{by definition of substitution} \tag{18}$$
$$\equiv \boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,(e^\div \;@\; \boldsymbol{\alpha}\;\boldsymbol{\lambda}\,\mathbf{x}.\,\mathbf{k}\;\mathbf{x}) \qquad \text{by Lemma 6.1} \tag{19}$$
$$\equiv \boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,(\boldsymbol{\lambda}\,\mathbf{x}.\,\mathbf{k}\;\mathbf{x})\;(e^\div \; \_ \; \mathbf{id}) \qquad \text{by } [\equiv\text{-Cont}] \tag{20}$$
$$= (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,(\boldsymbol{\lambda}\,\mathbf{x}.\,\mathbf{k}\;\mathbf{x})\;\mathbf{x})[(e^\div \; \_ \; \mathbf{id})/\mathbf{x}] \qquad \text{by substitution} \tag{21}$$
$$\equiv (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,\mathbf{k}\;\mathbf{x})[(e^\div \; \_ \; \mathbf{id})/\mathbf{x}] \qquad \text{by } \triangleright_\beta \tag{22}$$
$$= x^\div[(e^\div \; \_ \; \mathbf{id})/\mathbf{x}] \qquad \text{by definition of translation} \tag{23}$$

□

Lemma 6.3 (CPS^v is Type Preserving (Explicit Syntax)).

(1) *If* $\vdash \Gamma$ *then* $\vdash \Gamma^+$
(2) *If* $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash e^\div : A^\div$
(3) *If* $\Gamma \vdash A : \kappa$ *then* $\Gamma^+ \vdash A^+ : \kappa^+$
(4) *If* $\Gamma \vdash A : *$ *then* $\Gamma^+ \vdash A^\div : *^+$
(5) *If* $\Gamma \vdash \kappa : U$ *then* $\Gamma^+ \vdash \kappa^+ : U^+$

Proof. All cases are proven simultaneously by mutual induction on the typing derivation and well-formedness derivation. Part 4 follows easily by part 3 in every case, so we elide its proof. Most cases follow easily from the induction hypotheses.

Case [W-Def] $\vdash \Gamma, x = e : A$
We give the case for when A is a type; the case when A is a kind is similar.
We must show $\vdash \Gamma^+, \mathbf{x} = e^\div \; A^+ \; \mathbf{id} : A^+$.
It suffices to show that $\Gamma^+ \vdash e^\div \; A^+ \; \mathbf{id} : A^+$.
By part 2 of the induction hypothesis and definition of the translation, we know that $\Gamma^+ \vdash e^\div : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : *.\,(A^+ \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$, easily which implies the goal.

Case [Var] $\Gamma \vdash x : A$
We give the case for when A is a type; the case when A is a kind is simple since the translation on type variables is the identity.
We must show that $\Gamma^+ \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha} : *.\,\boldsymbol{\lambda}\,\mathbf{k} : A^+ \to \boldsymbol{\alpha}.\,\mathbf{k}\;\mathbf{x} : A^\div$
By the part 1 of the induction hypothesis, we know $\Gamma^+ \vdash \mathbf{x} : A^+$, which implies the goal.

Case [App] $\Gamma \vdash e_1\; e_2 : B[e_2/x]$.
We must show that

$$\Gamma^+ \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha} : *.\,\boldsymbol{\lambda}\,\mathbf{k} : (B^+[(e_2^\div \; A^+ \; \mathbf{id})/\mathbf{x}]) \to \boldsymbol{\alpha}. \qquad : (B[e_2/x])^\div$$
$$e_1^\div \; \boldsymbol{\alpha}\;(\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : A^+.\,B^\div.\,e_2^\div \;@\; \boldsymbol{\alpha}\;(\boldsymbol{\lambda}\,\mathbf{x} : A^+.\,(\mathbf{f}\;\mathbf{x})\;\boldsymbol{\alpha}\;\mathbf{k}))$$

Note that,

$$(B[e_2/x]) \tag{24}$$
$$\equiv B^{\div}[e_2^{\div}\ A^{+}\ \mathbf{id}/\mathbf{x}] \quad \text{by Lemma 6.2} \tag{25}$$
$$\equiv \Pi\ \boldsymbol{\alpha} : *.\ ((B^{+}[e_2^{\div}\ A^{+}\ \mathbf{id}/\mathbf{x}]) \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha} \quad \text{by translation} \tag{26}$$

Hence it suffices to show that
$\Gamma^{+}, \boldsymbol{\alpha} : *, \mathbf{k} : (B^{+}[(e_2^{\div}\ A^{+}\ \mathbf{id})/\mathbf{x}]) \to \boldsymbol{\alpha} \vdash e_1^{\div}\ \boldsymbol{\alpha}\ (\lambda\, \mathbf{f} : \Pi\, \mathbf{x} : A^{+}.\ B^{\div}.\ e_2^{\div}\ @\ \boldsymbol{\alpha}\ (\lambda\, \mathbf{x} : A^{+}.\ (\mathbf{f}\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k})) : \boldsymbol{\alpha}$
By part 2 of the induction hypothesis, we know that $\Gamma^{+} \vdash e_1^{\div} : \Pi\, \boldsymbol{\alpha} : *.\ ((\Pi\, \mathbf{x} : A^{+}.\ B^{\div}) \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$,
hence it suffices to show that
$\Gamma^{+}, \boldsymbol{\alpha} : *, \mathbf{k} : (B^{+}[(e_2^{\div}\ A^{+}\ \mathbf{id})/\mathbf{x}]) \to \boldsymbol{\alpha}, \mathbf{f} : \Pi\, \mathbf{x} : A^{+}.\ B^{\div} \vdash e_2^{\div}\ @\ \boldsymbol{\alpha}\ (\lambda\, \mathbf{x} : A^{+}.\ (\mathbf{f}\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k}) : \boldsymbol{\alpha}$
By [T-Cont], we must show
$\Gamma^{+}, \boldsymbol{\alpha} : *, \mathbf{k} : (B^{+}[(e_2^{\div}\ A^{+}\ \mathbf{id})/\mathbf{x}]) \to \boldsymbol{\alpha}, \mathbf{f} : \Pi\, \mathbf{x} : A^{+}.\ B^{\div}, \mathbf{x} = e_2^{\div}\ A^{+}\ \mathbf{id}, \vdash (\mathbf{f}\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k} : \boldsymbol{\alpha}$
Note that $\mathbf{f}\ \mathbf{x} : B^{\div}[\mathbf{x}/\mathbf{x}]$ and $B^{\div}[\mathbf{x}/\mathbf{x}] = \Pi\, \boldsymbol{\alpha} : *.\ (B^{+}[\mathbf{x}/\mathbf{x}]) \to \boldsymbol{\alpha} \to \boldsymbol{\alpha}$.
But $\mathbf{k} : (B^{+}[(e_2^{\div}\ A^{+}\ \mathbf{id})/\mathbf{x}]) \to \boldsymbol{\alpha}$.
Hence it suffices to show that $(B^{+}[\mathbf{x}/\mathbf{x}]) \equiv (B^{+}[(e_2^{\div}\ A^{+}\ \mathbf{id})/\mathbf{x}])$, which follows by δ reduction on $\mathbf{x}$ since we have $\mathbf{x} = e_2^{\div}\ A^{+}\ \mathbf{id}$ by our new typing rule [T-Cont].
Note that without our new typing rule, we would be here stuck. However, thanks to [T-Cont], we have the equality that $\mathbf{x} = e_2^{\div}\ A^{+}\ \mathbf{id}$, and we are able to complete the proof. □

Theorem 6.4 (CPS^{v} is Type Preserving (PTS syntax)). *If* $\Gamma \vdash e : A$ *then* $\Gamma^{+} \vdash \text{cps}\ [\![e]\!] : \text{cpsT}\ [\![A]\!]$.

6.2 Proof of Correctness for CPS^{v}

To prove correctness of separate compilation for CPS^{v}, we follow the same recipe as in Section 5.2. We use the same cross-language relation $\approx$ on values of ground type. However, note that in CBV we should only link with values, so we restrict closing substitutions γ to values and use the value translation on substitutions γ^{+}. The proofs follow exactly the same structure as in Section 5.2.

Theorem 6.5 (Separate Compilation Correctness). *If* $\Gamma \vdash e : A$ *where* A *is ground, and* $\gamma(e) \rhd^{\star} v$ *then* $\gamma^{+}(e^{\div})\ A^{+}\ \mathbf{id} \rhd^{\star} \mathbf{v}$ *and* $v \approx \mathbf{v}$.

Corollary 6.6 (Whole-Program Compiler Correctness). *If* $\vdash e : A$ *and* $e \rhd^{\star} v$ *then* $e^{\div}\ A^{+}\ \mathbf{id} \rhd^{\star} \mathbf{v}$ *and* $v \approx \mathbf{v}$.

7 DISCUSSION AND FUTURE WORK

Dependent Sums. In addition to showing that the traditional double-negation CPS for Σ types is not type preserving, Barthe and Uustalu [2002] demonstrate an impossibility result for CPS and sums with dependent case analysis. They prove that no type-correct CPS translation can exist for sums with dependent case. However, careful inspection of their proof reveals that this impossibility result does not apply to our CPS translation. Furthermore, we can define a CPS translation for dependent case that appears to be type preserving, although we do not give a formal proof.

The impossibility result by Barthe and Uustalu [2002] relies on the ability to implement `call/cc` via the CPS translation. Assuming there is a type-preserving CPS translation, they construct a model of CC extended with `call/cc` and sum types (CCΔ+) in CC with sum types (CC+). Since CC+ is consistent, this model proves that CCΔ+ is consistent. However, it is known that CCΔ+ is inconsistent [Coquand 1989]. Therefore, the type-preserving CPS translation of CCΔ+ cannot exist.

Their proof is valid; however, it is well known that the polymorphic answer type CPS translation that we use cannot implement `call/cc` [Ahmed and Blume 2011]. Therefore, our translation does not give a model of CCΔ+ in CC+.

We go further and conjecture a positive result, that our CPS translation is type preserving for dependent case. The typing rule for dependent case is the following.

$$\frac{\Gamma, y : A + B \vdash A' : * \qquad \Gamma, x : A \vdash e_1 : A'[\mathsf{inj}_1\ x/y] \qquad \Gamma, x : B \vdash e_2 : A'[\mathsf{inj}_2\ x/y]}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ x.\ e_1 \mid x.\ e_2 : A'[e/y]}$$

We would extend our CPS^n with the following rule.

$$(\mathsf{case}\ e\ \mathsf{of}\ x.\ e_1 \mid x.\ e_2)^{\dot\div} = \boldsymbol{\lambda}\ \boldsymbol{\alpha} : *.\ \boldsymbol{\lambda}\ \mathbf{k} : A'^{+}[e^{\dot\div}/\mathbf{y}] \to \boldsymbol{\alpha}.\ e^{\dot\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\ \mathbf{y} : A^{+} + B^{+}.\ \mathbf{case}\ \mathbf{y}\ \mathbf{of}\ \mathbf{x}.\ (e_1^{\dot\div}\ \boldsymbol{\alpha}\ \mathbf{k}) \mid \mathbf{x}.\ (e_2^{\dot\div}\ \boldsymbol{\alpha}\ \mathbf{k}))$$

Focusing on the first branch of the case above, note that $e_1^{\dot\div}\ \boldsymbol{\alpha} : (A'^{+}[(\mathsf{inj}_1\ x)^{\dot\div}/\mathbf{y}] \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$, however $\mathbf{k} : A'^{+}[e^{\dot\div}/\mathbf{y}] \to \boldsymbol{\alpha}$. We need to show that $e^{\dot\div} \equiv (\mathsf{inj}_1\ x)^{\dot\div}$, similar to the problem with the second projection of dependent pairs. This time, however, to show $e^{\dot\div} \equiv (\mathsf{inj}_1\ x)^{\dot\div}$ we need to reason about the flow of the underlying value of $e^{\dot\div}$ into $\mathbf{y}$ and also about the relationship between $\mathbf{y}$ and $\mathbf{x}$. Specifically, we need to first use our new [T-Cont] rule, which allows us to assume $\mathbf{y} = e^{\dot\div}\ \boldsymbol{\alpha}\ \mathbf{id}$. Next, we need to know that since the case analysis is on the value $\mathbf{y}$, in the first branch $\mathbf{y} \equiv \mathbf{inj}_1\ \mathbf{x}$ (and similarly for the other branch), but the problem is that our existing typing rule for dependent case does not let us assume that. Nonetheless, past work on dependent elimination in Agda suggests that we can modify the typing rule for dependent case to have the equality $e \equiv \mathsf{inj}_1\ x$ while typing the first branch, and similarly for the second branch [Cockx et al. 2016]. With this additional equality in hand, some simple computation with the translation of $(\mathsf{inj}_1\ x)$ gives us the desired equivalence. Hence, with the aforementioned modification to the typing rule for dependent case in our target language CC^k, we can type check the above translation of dependent case. The part of the proof that we have not yet carried out is showing the consistency of our target language CC^k once it is extended with this modified typing rule for dependent case.

The Calculus of Inductive Constructions. We haven't yet completed the above proof for sums with dependent elimination because ultimately we want to extend our translation to the Calculus of Inductive Constructions (CIC). Inductive types have a similar structure to sums with dependent case, but much more involved typing rules. We believe that the sketch presented for dependent case on sums extends to general inductive types and intend to extend our translation in the future. However, CIC presents several other challenges.

The first is to handle the infinite hierarchy of universes. In this work, we have a single impredicative universe $*$, and the locally polymorphic answer type $\boldsymbol{\alpha}$ lives in that universe. With an infinite hierarchy, it is not clear what the universe of $\boldsymbol{\alpha}$ should be. Furthermore, our work relies on impredicativity in $*$. We can only use impredicativity at one universe level and the locally polymorphic answer-type translation has not been studied in the context of predicative polymorphism, so it's unclear how to adapt our translation to the infinite predicative hierarchy.

The right solution for handling universes seems to be universe polymorphism [Sozeau and Tabareau 2014]. Since the type is provided by the calling context, it seems sensible that the calling context should also provide the universe. We could modify the type translation to be $\mathbf{\Pi}\ \ell : \mathbf{Level}.\ \mathbf{\Pi}\ \boldsymbol{\alpha} : \mathbf{Type}_\ell.\ (A^{+} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$. However, it is not clear how universe polymorphism would affect the rest of the CPS translation.

Finally, CIC also features guarded recursive functions. As the guard condition is purely syntactic, it seems likely to be disrupted by CPS translation. We either need to compile the guard condition to something more semantic, like sized types, before the CPS translation, or adapt the guard condition to the new CPS syntax.

CPS vs ANF. ANF [Flanagan et al. 1993], which is based on a structured use of let, is a popular alternative to CPS for compiler intermediate languages. It is natural to wonder whether ANF would be easier than CPS in CC, especially given the complexity of CPS in CC and the fact that our proof

of consistency performs a translation similar to the inverse CPS translation of Flanagan et al. [1993]. We conjecture that, while the ANF translation on its own may be simpler owing to the additional expressivity[6] of **let**, using ANF in practice will not be significantly simpler than using our CPS translation. Kennedy [2007] points out that ANF is not preserved under β-reduction, and that ANF in the presence of conditional expressions can result in considerable code duplication; neither problem arises in CPS. Recently, Maurer et al. [2017] formalize a notion of *join-points* to solve the latter problem, observing that this simplifies certain optimizations that are difficult in CPS. We may be able to adapt this work to the dependently typed setting, but we conjecture that formalizing dependent join-points will require an analog of [≡-Cont], and ANF will complicate β-reduction.

Implementing call/cc, or Predicative or Non-parametric Models. Our work relies on parametricity to recover type preservation, but avoiding parametricity has certain benefits, such as the ability to implement restricted forms of `call/cc`, or using predicative or non-parametric models of dependent type theory. We use parametricity as a type-based mechanism to enforce a condition known as *naturality*—*i.e.*, the equivalence given by [≡-Cont]—but naturality can be defined in non-parametric or even untyped settings. For instance, Thielecke [2003] gives a definition of naturality in terms of a delimited continuations: a term M is natural if $M \equiv \%M$ (where $\%$ is the continuation delimiter `prompt`). Using parametricity to enforce naturality limits the scope of our CPS translation. We cannot use our translation to implement `call/cc`. This restriction is unnecessary in principle—restricted uses of `call/cc` are consistent with dependent type theory [Herbelin 2012]. Our translation requires impredicativity and parametricity in the target language type theory, and these two features are not always admitted in type theory. For instance, Agda is predicative, and Boulier et al. [2017] give some non-parametric models of type theory. Thielecke [2003] studies how to recover properties like naturality in the presence of `call/cc` using a source language with type-and-effect system, although using a CPS translation based on answer-type polymorphism. As future work, investigating how to extend that work to dependent type theory may let us broaden the scope of our CPS translation.

Weak Normalization Implies Strong Normalization. In this paper, we only CPS translate *terms*, *i.e.*, run-time expressions. However, other work [Barthe et al. 2001] studies *pervasive* translation of PTSs. A pervasive CPS translation internalizes evaluation order for all expressions: terms, types, and kinds. A pervasive CPS translation has been used to give a partial solution to the Barendregt-Geuvers-Klop conjecture [Geuvers 1993] which essentially states that every weakly normalizing PTS is also strongly normalizing. The conjecture has been solved for non-dependent PTSs, but the solution for dependent PTSs remains open. As future work, we intend to extend our work to a pervasive translation. This may allow us to make progress on the Barendregt-Geuvers-Klop conjecture. However, as our translation requires additional axioms, it's not clear to what class of dependent PTSs the result might apply.

8 RELATED WORK

Type-Preserving Compilation of Dependent Types. The challenges of extending CPS translations to dependently typed languages were first pointed out by Barthe et al. [1999]. As discussed in Section 5.1, a difficulty arises when working in the domain-full lambda calculus with a *typed* equivalence: we cannot stage the proofs of compositionality, coherence, and type preservation. Barthe et al. solve this problem using domain-free lambda abstractions, and successfully define a type-preserving call-by-name CPS translation for CC (without Σ types). Unfortunately, type

[6]Unfortunately, we lack a formal system for discussing the relative expressiveness of type features along the same lines as macro expressibility used to discuss expressiveness of runtime features [Felleisen 1991]. We therefore appeal to the reader's intuition.

checking becomes undecidable in the domain-free calculus. We avoid the circularity by using an *untyped* equivalence, which allows us to use a *domain-full* calculus, eliminating one source of undecidability. Recall from Section 4.1 that since we provide a model of our target language CC^k in the extensional CC, we do not yet have a proof that type checking in CC^k is decidable.

Other approaches to type-preserving compilation for dependent types avoid the difficulties we've seen with CPS by avoiding *full spectrum* dependent types, *i.e.*, Σ and Π with no explicit distinctions between terms and types. (Full spectrum dependent types support code reuse across terms and types, and avoid the need for programmers to marshall between different term and type representations.) Shao et al. [2005] develop type-preserving CPS and closure-conversion passes for a language that uses CIC as an extrinsic type system, for use in developing certified binaries. In their language, types *cannot* depend on run-time expressions. Instead, certain expressions have a second, type-level representation, and CIC specifications and proofs can be written for these type-level representations. One motivation for this design was exactly the problem with CPS we have solved. Chen et al. [2010] develop a type-preserving compiler from Fine, an ML-like language with refinement types instead of full spectrum dependent types, to DCIL, a typed variant of the .NET Common Intermediate Language with type-level computation. The type system in DCIL was used to encode security and verification conditions, also supporting certified binaries.

Control Operators and Dependent Types. There is a line of work combining dependent types and control operators. Herbelin [2005] shows that unrestricted use of `call/cc` and `throw` in a language with Σ types and equality leads to an inconsistent system. The inconsistency is caused by type dependency on terms involving control effects. Herbelin [2012] solves the inconsistency by constraining types to depend only on *negative-elimination-free (NEF)* terms, which are free of effects. This restriction makes dependent types compatible with classical reasoning enabled by the control operators. Our CPS translation doesn't allow implementing control operators, but this line of work leads us to suspect that there exists a less restricted translation that preserves typing.

Recent work by Miquey [2017] uses the NEF restriction to soundly extend the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [2000], a computational classic calculus, with dependent types. He then extends the language with delimited continuations, and defines a type-preserving CPS to a standard intuitionistic dependent type theory. By comparison, our source language CC is effect-free, therefore we do not need the NEF condition to restrict dependency. Our use of the identity function serves a similar role to their delimited continuations—allowing local evaluation of a CPS'd computation.

Miquey [2017] makes an interesting observation that certain evaluation contexts in dependently typed languages are not well typed, and we conjecture that our work offers a solution. As the $\bar{\lambda}\mu\tilde{\mu}$-calculus has explicit evaluation contexts in the syntax, he observes that subject reduction fails, and calls this "desynchronization of typing with respect to the execution." We can see this problem in CC evaluation contexts as well, although evaluation contexts are purely meta-theoretic features and do not exist in the syntax of CC. Consider a well-typed CBN context representing a function application, $E\ e : (\Gamma' \vdash A') \Rightarrow (\Gamma \vdash B[e/x])$, which says that when we plug a term of type A' into the hole of E, this context yields a term of type $B[e/x]$. In general, contexts can capture variables, so we include in the type the environment Γ' of the hole and remaining free variables Γ after plugging a term. This works for a CBN application context, but in CBV we have an additional evaluation context for application, $v\ E$, which evaluates the argument. The type of $v\ E$ now depends on the context E, not on a term, so it is not clear how to proceed: $v\ E : (\Gamma' \vdash A') \Rightarrow (\Gamma \vdash B[???/x])$. In both CBN and CBV, the same problem arises in the evaluation context for the second projection: $\mathsf{snd}\ E : (\Gamma' \vdash A') \Rightarrow (\Gamma \vdash B[\mathsf{fst}\ ???/x])$. As our CPS translation produces well-typed continuations from these apparently ill-typed contexts, it should be possible to design a source syntax for well-typed CC evaluation contexts that is realized by our translation.

Effects and Dependent Types. Pédrot and Tabareau [2017] define the *weaning translation*, a monadic translation for adding effects to dependent type theory. The weaning translation allows us to represent self-algebraic monads, *i.e.*, monads whose algebras' universe itself forms an algebra, such as exception and non-determinism monads. However, it does not apply to the standard continuation monad, which is not self-algebraic. The paper extends the translation to inductive types, with a restricted form of dependent elimination. Full dependent elimination can be implemented only for terms whose type is first-order, and this comes at the cost of inconsistency, although one can recover consistency by requiring that every inductive term be parametric. Our translation does not lead to inconsistency, and requires no restrictions on the type to be translated. However, our translation appears to impose the self-algebraic structure on our computation types, and our use of parametricity to cast computations to values is similar to their parametricity restriction.

Multi-language Semantics. A full spectrum dependently typed language resembles a *multi-language system* [Matthews and Findler 2007] between a (compile time) type language and a (run time) term language. In CC, the two languages happen to be the same, but the distinction between them is made clear via CPS. We recover type preservation for CPS translations by defining interoperability between the CPS'd terms and the direct-style types. To define interoperability, we allow types to run a CPS'd term to a value by supplying the halt, or identity, continuation. This is exactly the interoperability semantics defined by Ahmed and Blume [2011] between a CPS and direct-style language. Their work also makes use of answer-type polymorphism to enforce naturality. A similar strategy appears in the multi-language semantics given by Patterson et al. [2017] for interoperability between a direct-style functional language and a continuation-based assembly language. Their work does not use answer-type polymorphism, but enforces naturality by augmenting the type system to track return addresses and adding a special halt instruction to interpret assembly components as values.

Applications of the Polymorphic Answer Type. The polymorphic answer type has proven useful in other work on CPS translations as well. Thielecke [2003] studies the connection between control effects and answer-type polymorphism in a call-by-value language with `call/cc`. He defines an effect system that tells us whether we can regard a term as a pure term. Then he shows pure terms use their continuation linearly, and that their answer type is polymorphic. Using the latter property, he proves that enclosing a pure term with a control delimiter does not affect its meaning, *i.e.*, *naturality*. Thielecke [2004] extends this study to the call-by-name setting, showing that answer-type polymorphism holds not only for pure functions, but also for effectful functions.

Ahmed and Blume [2011] give a full-abstraction proof of a call-by-value CPS translation that uses polymorphic answer types. Full abstraction is an important property for secure compilation, which ensures that a translation converts two terms that are indistinguishable in any source context into two terms that are indistinguishable in any target context. Their proof relies on exactly the free theorem we proved in Section 4.1. We conjecture therefore that our CPS translation is fully abstract.

Internalizing Parametricity. Our work internalizes a specific free theorem, but ongoing work focuses on how to internalize parametricity more generally in a dependent type theory. Krishnaswami and Dreyer [2013] develop a technique for adding new rules to the extensional CC. They present a logical relation for terms that are not syntactically well typed, but are semantically well behaved and equivalent at a particular type. Using this logical relation, they prove the consistency of several extensions to extensional CC, including sum types, dependent records, and natural numbers. Bernardy et al. [2012]; Keller and Lasson [2012] give translations from one dependent type theory into another that yield a parametric model of the original theory. These essentially encode the

logical relation in the target type theory, similar to the approach we took in Section 4.1 (Consistency of CC^k). Recent work by Nuyts et al. [2017] develops a theory that internalizes parametricity, including the important concept of *identity extension*, and gives a thorough comparison of the literature. By building a target language based on one of these systems, it's possible that we could eliminate the rule [≡-Cont] as an axiom and instead derive it internally.

Verified Compilation. CertiCoq is a mechanically verified optimizing compiler currently under development [Anand et al. 2017]. It compiles Coq's Gallina to CompCert's Clight. As mentioned in Section 1, CertiCoq is *not* a type-preserving compiler: it erases types before CPS conversion, optimizations, and closure conversion. CertiCoq's current compiler correctness theorem aims to prove separate compilation correctness. We have proved similar separate-compilation correctness theorems, Theorem 5.7 and Theorem 6.5, but our typed target language can statically enforce these guarantees via type checking. Our theorems differ in a few key aspects. First, we only show correctness for the CPS translation of CC. Second, our relation on ground terms is impoverished; we only prove that booleans are related, while CertiCoq has an abstract relation that is preserved on all ground types. Finally, our long-term goals are different from CertiCoq's current goal of correct separate compilation. We are interested in type-preserving compilation of Coq as a means of supporting correct and secure (fully abstract) compilation even when the compiled code is linked with code compiled from different, possibly effectful, languages. By picking the right type translation for each phase of the compiler, we wish to use the target-level type specification to—statically, or dynamically via gradual typing—rule out unsafe or insecure interactions with target code. In our CPS translation, the polymorphic answer type does the work to rule out interaction with computations that have control effects. Therefore, even if we could encode `call/cc` in the target language (which is type safe when restricted to Herbelin's negative-elimination-free fragment), it would not interoperate with the output of our compiler.

ACKNOWLEDGMENTS

We gratefully acknowledge the valuable feedback provided by Greg Morrisett, Karl Crary, and the anonymous reviewers. 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 a Northeastern University Scholars' Independent Research Fellowship. This work was, in part, supported by 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 the National Science Foundation.

REFERENCES

Martín Abadi. 1998. Protection in Programming-language Translations. In *International Colloquium on Automata, Languages, and Programming*. https://doi.org/10.1007/bfb0055109

Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. https://doi.org/10.4230/LIPIcs.SNAPL.2015.15

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1411203.1411227

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

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 *The International Workshop on Coq for Programming Languages (CoqPL)*. http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf

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

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

Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. 2001. Weak Normalization Implies Strong Normalization in a Class of Non-dependent Pure Type Systems. *Theoretical Computer Science* 269, 1-2 (Oct. 2001). https://doi.org/10.1016/s0304-3975(01)00012-3

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

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

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

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

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

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-Preserving CPS Translation of Σ and Π Types Is Not Not Possible (Supplementary Materials). (Oct. 2017). https://williamjbowman.com/resources/cps-sigma.tar.gz

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

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers As Equivalences: Proof-relevant Unification of Dependently Typed Data. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2951913.2951917

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

Thierry Coquand. 1989. *Metamathematical Investigations of a Calculus of Constructions*. Ph.D. Dissertation. INRIA. https://hal.inria.fr/inria-00075471

Pierre-louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/357766.351262

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17, 1-3 (Dec. 1991). https://doi.org/10.1016/0167-6423(91)90036-W

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/155090.155113

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-evariste Dagand, Pierre-yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2480359.2429114

Jan Herman Geuvers. 1993. *Logics and Type Systems*. Ph.D. Dissertation. University of Nijmegen. http://www.ru.nl/publish/pages/682191/geuvers_jh.pdf

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

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

Hugo Herbelin. 2012. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/lics.2012.47

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

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

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

Andrew Kennedy. 2006. Securing the .NET Programming Model. *Theoretical Computer Science* 364, 3 (Nov. 2006). https://doi.org/10.1016/j.tcs.2006.08.014

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1291220.1291179

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

Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1111037.1111042

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

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1190216.1190220

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without Continuations. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062380

Étienne Miquey. 2017. A Classical Sequent Calculus with Dependent Types. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-54434-1_29

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

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

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

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Transactions on Programming Languages and Systems* 37, 2, Article 6 (April 2015). https://doi.org/10.1145/2699503

Daniel Patterson and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. https://doi.org/10.4230/LIPIcs.SNAPL.2017.12

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)*. http://www.ccs.neu.edu/home/amal/papers/funtal.pdf

Pierre-marie Pédrot and Nicolas Tabareau. 2017. An Effectful Way to Eliminate Addiction to Dependence. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/lics.2017.8005113

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

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

Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *International Conference on Interactive Theorem Proving (ITP)*. https://doi.org/10.1007/978-3-319-08970-6_32

The Coq Development Team. 2017. The Coq Proof Assistant Reference Manual. https://coq.inria.fr/doc/Reference-Manual006.html

Hayo Thielecke. 2003. From Control Effects to Typed Continuation Passing. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/604131.604144

Hayo Thielecke. 2004. Answer Type Polymorphism in Call-by-name Continuation Passing. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-540-24725-8_20