Compiling Dependent Types Without Continuations

WILLIAM J. BOWMAN, University of British Columbia, CA
AMAL AHMED, Northeastern University, USA

Programmers rely on dependently typed languages such as Coq to machine-verify high-assurance software, but these languages provide no guarantees after compiling nor when linking after compilation. We could provide guarantees after compiling and linking by building *type-preserving compilers*, which preserve specifications encoded in types and use type checking to verify code after compilation and to ensure safety when linking with external code. Unfortunately, standard type-preserving translations do not scale to dependently typed languages for two key reasons: assumptions valid in simpler type systems no longer hold, and dependent type systems are highly sensitive to changes in the syntax of programs, which are common during compilation.

We extend the A-normal form (ANF) translation with join-point optimization—a standard translation for making control flow explicit—to the Extended Calculus of Constructions (ECC) with dependent elimination of booleans, a representative subset of Coq. We prove type preservation and correctness of separate compilation from ECC to an ANF-restricted variant of ECC with a machine semantics. This is the first ANF translation for dependent types and, unlike related translations, supports the infinite universe hierarchy, and does so without relying on parametricity or impredicativity. Our work provides general insights into dependent-type-preservation and combining effects with dependent types.

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

1 INTRODUCTION

Dependently typed languages such as Coq, Agda, Idris, and F* allow programmers to write full-functional specifications for their programs (or program components), implement the program, and prove that the program meets its specification. These languages, and Coq in particular, have been widely used to build formally verified high-assurance software including the CompCert C compiler [Leroy 2009], the CertiKOS operating system kernel [Gu et al. 2015, 2016], and cryptographic primitives [Appel 2015] and protocols [Barthe et al. 2009].

Unfortunately, even these machine-verified programs can contain errors when executed due to errors introduced during compilation and linking. For example, suppose we have a program component S that we have written and proven correct in a *source* language like Coq. To execute S, we first compile S from Coq to a component T in OCaml. If the compiler from Coq to OCaml introduces an error, we say that a *miscompilation error* occurs. Now T contains an error despite S being verified. But this is not the end of the story, since S and T are not whole programs, but components that rely on other code. Next, T will be linked with some external (*i.e.*, not implemented and verified in Coq) code C to form the whole program P. If C violates the original specification of S, then we say a *linking error* occurs and P contains safety, security, or correctness errors.

A verified compiler is one that prevents miscompilation errors, since it is proven to preserve the run-time behavior of a program, but it cannot prevent linking errors. In our example above, note that linking errors can occur *even if S is compiled with a verified compiler*, since the external code we link with, C, is outside of the control of either the source language or the verified compiler. Ongoing work on CertiCoq [Anand et al. 2017] seeks to develop a *verified* compiler for Coq, but it cannot rule out linking with unsafe target code. One can develop simple examples in Coq that, once

*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 fonts are distinguishable in black-and-white, but the paper is easier to read when viewed in color.

Authors' addresses: William J. Bowman, University of British Columbia, CA, wjb@williamjbowman.com; Amal Ahmed, Northeastern University, USA, amal@ccs.neu.edu.

compiled to OCaml and linked with an unverified OCaml component, *jump to an arbitrary location in memory*—despite the Coq component being proven memory safe with a machine-checked proof.

To rule out both miscompilation and linking errors, we could combine compiler verification with *type-preserving compilation*. A type-preserving compiler preserves types, representing specifications, into a typed intermediate language (IL). The IL uses type checking at link time to enforce specifications when linking with external code, essentially implementing proof-carrying code [Necula 1997]. Type-preserving compilation is already used in this way to target the Java Virtual Machine and the .NET Common Intermediate Language. After linking in the IL, we have a whole program, so it is sufficient to erase types and use verified compilation to machine code. To support safe linking with untyped code, we could use gradual typing to enforce safety at the boundary between the typed IL and untyped components [Ahmed 2015]. Applied to Coq, this technique would provide significant confidence that the executable program P is as correct as the verified program S.

Unfortunately, type-preserving compilation for dependently typed languages is challenging, particularly for translations that make control flow explicit. Supporting each new *feature of dependency*—the core features of dependent types used in practice—has come with some significant limitation. Barthe et al. [1999] were able to extend the standard type-preserving CPS translation to support Π types, the most basic dependent type, but only at the expense of decidable type checking. Barthe and Uustalu [2002] tried to extend this further, to support Σ types (another basic and necessary feature), but ran into problems with consistency. Bowman et al. [2018] manage to support both Π and Σ while maintaining consistency, but cannot scale to higher universes (another basic and necessary feature), and rely on parametricity and impredicativity (strong axioms that are orthogonal to dependent type theory and thus restrict which source programs can be compiled).

To have a hope of being practical, *i.e.*, of scaling to a language such as Coq, we need a theory for dependent-type preserving translations that supports the features of dependency found in Coq—Π, Σ, higher universes, dependent case analysis, and all the axioms that Coq users might rely on in their components. (Inductive types can be encoded using strictly positive recursive types plus the above features; as recursive types introduce no new dependencies, they should not introduce new problems for type preservation.) So now the question is: can we develop a type-preserving translation in the presence of all the above features?

The answer is yes. In this paper, we develop a type-preserving translation from ECC, the Extended Calculus of Constructions (ECC) with dependent elimination of booleans, to A-normal form (ANF), a compiler intermediate form for making control flow explicit and performing optimizations [Flanagan et al. 1993; Sabry and Felleisen 1992]. ECC represents a significant subset of Coq as it contains all core features of dependency discussed earlier. The focus of this work is supporting higher universes and avoiding parametricity and impredicativity, which are primary limitations in prior work [Bowman et al. 2018; Cong and Asai 2018a]. This work provides substantial evidence that dependent-type preservation can, in theory, scale to the dependently typed languages used for high-assurance software.

Our translation targets ECC^A, which is essentially an ANF-restricted variant of ECC. ECC^A features a machine-like semantics for evaluating ANF terms, and we prove correctness of separate compilation with respect to this machine semantics. To support dependent elimination of booleans and join-point optimization, ECC^A requires three extensions. We provide a model of ECC^A in CIC with equivalence reflection[1] (eCIC), and formalize the key proofs in Coq (included in [2]). Unlike prior type-preserving translations, the translation is completely standard, and the type system does not rely on parametricity or impredicativity. This ensures that the translation works for existing

[1] While our model relies on equivalence reflection, we conjecture ECC^A itself does not.

[2] https://www.williamjbowman.com/downloads/anf-sigma-techrpt.pdf

dependently typed languages, that we can reuse existing work on ANF translation, and that the work scales to many extensions that are compatible with dependent type theory.

Our ANF translation is useful as a compiler pass, but also provides insights into dependent-type preserving translations. In particular, dependent-type preserving translations must convert dependencies on arbitrary terms into a series of dependencies on *machine steps*, and the encoding of machine steps must avoid new axioms. We spend the rest of this paper decompressing this sentence in the context of ANF, and explicitly apply these insights to related translations and areas of research in Section 7. This paper includes key definitions and proof cases; extended figures and proofs are available in [3].

2 MAIN IDEAS

ANF 101. A-normal form (ANF)[4] is a syntactic form that makes control flow explicit in the syntax of a program [Flanagan et al. 1993; Sabry and Felleisen 1992]. ANF encodes *computation* (*e.g.*, reducing an expression to a value) as the sequencing of simple intermediate computations with *let* expressions. To reduce e to a value in a high-level language, we need to describe the evaluation order and control flow of each language primitive. For example, if e is an application $\mathrm{e}_1\ \mathrm{e}_2$ and we want a call-by-value semantics, we say the language first evaluates e_1 to a value, then evaluates e_2 to a value, then performs the function application. ANF makes this explicit in the syntax by decomposing e into the series of primitive computations that the machine must execute, sequenced by **let**.

Roughly, we can think of the ANF translation $[\![\mathrm{e}_1\ \mathrm{e}_2]\!]$ as $\mathbf{let}\ \mathrm{x}_1 = [\![\mathrm{e}_1]\!], \mathrm{x}_2 = [\![\mathrm{e}_2]\!]\ \mathbf{in}\ \mathrm{x}_1\ \mathrm{x}_2$. This rough translation is only ANF if e_1 and e_2 are values or primitive computations, but e_1 and e_2 could be deeply nested expressions in the source language. In general, the ANF translation reassociates all the intermediate computations from $[\![\mathrm{e}_1]\!]$ and $[\![\mathrm{e}_2]\!]$ so there are no nested **let** expressions, as shown below.

$$
\begin{aligned}
[\![\mathrm{e}_1\ \mathrm{e}_2]\!] = \mathbf{let}\ & \mathrm{x}_{1_0} = \mathrm{N}_{1_0}, ..., \mathrm{x}_{1_n} = \mathrm{N}_{1_n}, \mathrm{x}_1 = \mathrm{N}_1 \\
& \mathrm{x}_{2_0} = \mathrm{N}_{2_0}, ..., \mathrm{x}_{2_n} = \mathrm{N}_{2_n}, \mathrm{x}_2 = \mathrm{N}_2 \\
& \mathbf{in}\ (\mathrm{x}_1\ \mathrm{x}_2) \\
\text{where } [\![\mathrm{e}_1]\!] = (\mathbf{let}\ & \mathrm{x}_{1_0} = \mathrm{N}_{1_0}, ..., \mathrm{x}_{1_n} = \mathrm{N}_{1_n}\ \mathbf{in}\ \mathrm{N}_1) \\
[\![\mathrm{e}_2]\!] = (\mathbf{let}\ & \mathrm{x}_{2_0} = \mathrm{N}_{2_0}, ..., \mathrm{x}_{2_n} = \mathrm{N}_{2_n}\ \mathbf{in}\ \mathrm{N}_2)
\end{aligned}
$$

Once in ANF, it is simple to formalize a machine semantics to implement evaluation. Each **let**-bound computation $\mathrm{x}_i = \mathrm{N}_i$ is some primitive *machine step*, performing the computation N_i and binding the value to x_i. The machine proceeds by always reducing the left-most machine-step, which will be a primitive operation with values for operands. For a lazy semantics, we can instead delay each machine step and begin forcing the inner-most body (right-most expression).

Why ANF Translation Disrupts Typing and How to Fix it. The problem with dependent-type preservation for ANF has little to do with ANF itself, and everything to do with dependent types. Certain transformations which *ought* to be fine (the transformed term reduces to the original), aren't because the type theory is so beholden to details of syntax. This is essentially the problem of *commutative cuts* in type theory [Boutillier 2012; Herbelin 2009]. Transformations that change the structure of a program, the syntax, can disrupts *dependencies*. By dependency, we mean an expression e' whose type and evaluation depends on a sub-expression e. We call a sub-expression such as e *depended upon*. These dependencies occurs in dependent elimination forms, such as

[3] https://www.williamjbowman.com/downloads/anf-sigma-techrpt.pdf

[4] The *A* in *A-normal form* has no further meaning. The original work observes that the CPS/optimization/un-CPS process could be seen as a set *A* of source reductions. The *A-normal form* is the normal form resulting from applying the *A-reductions* until a normal form is reached.

application, projection, and dependent if expressions. Transforming a dependent elimination, via ANF translation or otherwise, can disrupt dependencies.

For example, the second projection of a dependent pair $e : \Sigma\, x : A.\, B$ is typed as follows:

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

Notice that the *depended upon* sub-expression e is copied into the type, indicated by the solid line arrow. If we transform the expression $\mathsf{snd}\, e$, we can easily change the type. For instance, suppose we just want to let-bind e before we project from it. This transformation is given below with a few suggestive type annotations.

$$\mathsf{let}\, y = e : \Sigma\, x : A.\, B\, \mathsf{in}\, \mathsf{snd}\, y : B[x := \mathsf{fst}\, y]$$

The type has changed from $B[x := \mathsf{fst}\, e]$ to $B[x := \mathsf{fst}\, y]$, merely because we changed the syntax. This transformation is only type preserving if $e \equiv y$, *i.e.*, if e is *provably* equivalent to y, in the type B.

In this simple case, we can fix the problem by using the following standard dependent typing rule for let.

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

This typing rule records the equality from the term in the type by substitution. Now, in the type B, $y \equiv e$ because $y = e$, through substitution. This is completely standard, but it doesn't solve the problem in general. Suppose instead we have the complex expression $f\ (\mathsf{snd}\, e) : C$, and we want to let-bind all intermediate computations (which, incidentally, ANF does).

$$\begin{array}{l}\mathsf{let}\, y = e : \Sigma\, x : A.\, B \\ \quad z = \mathsf{snd}\, y : B[x := \mathsf{fst}\, y]\, \mathsf{in} \\ \quad f\ z\end{array} \qquad \text{where } f : (B[x := \mathsf{fst}\, e]) \to C$$

This is not well typed, even with the dependent-let rule. The problem now is the dependent elimination $\mathsf{snd}\, y$ is let-bound and then used, instead of being returned as the body of the let expression as in the earlier example. This means the equality $y = e$ is missing, but is needed to type check this term (indicated by the dotted line arrow). The type derivation fails, as follows.

$$\dfrac{\dfrac{\dfrac{}{y : \Sigma\, x : A.\, B \vdash \mathsf{snd}\, y : B[x := \mathsf{fst}\, y]} \qquad \dfrac{\text{fails: } B[x := \mathsf{fst}\, y] \neq B[x := \mathsf{fst}\, e]}{y : \Sigma\, x : A.\, B, z : B[x := \mathsf{fst}\, y] \vdash f\ z : C}}{y : \Sigma\, x : A.\, B \vdash \mathsf{let}\, z = \mathsf{snd}\, y\, \mathsf{in}\, f\ z : C[z := \mathsf{snd}\, y]}}{\vdash \mathsf{let}\, y = e\, \mathsf{in}\, \mathsf{let}\, z = \mathsf{snd}\, y\, \mathsf{in}\, f\ z : C[z := \mathsf{snd}\, y][y := e]}$$

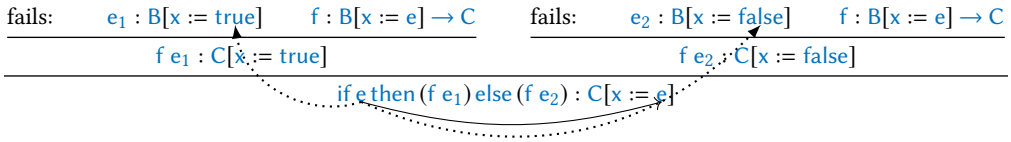
This fails since f expects $z : B[x := \mathsf{fst}\, e]$, but is applied to $z : B[x := \mathsf{fst}\, y]$. This wouldn't be a problem if we waited to apply f to the entire let expression, since the equality $y = e$ is essentially baked into the return type of the let. But if we want to avoid nested expressions, then this is too little too late. We really needed $y = e$ when checking the type of $\mathsf{snd}\, y$.

The typing rule essentially forgets that, by the time $\mathsf{snd}\, y$ happens, the machine will have performed the step of computation $y = e$, and it *ought* to be safe to assume that $y = e$ in the types. When type checking in linearized machine languages, like in ANF, we need to record these machine steps as we go *up* the typing derivation.

The above explanation applies to all dependent eliminations of negative types, such as Π and Σ. An analogous problem occurs with dependent elimination of positives types, such as booleans. For example, consider the following two terms, an if statement and f. We give part of the typing derivation for the if statement to remind the reader of the dependent types of the branches.

$$\frac{\vdash e_1 : B[x := \mathsf{true}] \qquad \vdash e_2 : B[x := \mathsf{false}]}{\vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B[x := e]} \qquad f : B[x := e] \to C$$

The expression $f\ (\text{if}\ e\ \text{then}\ e_1\ \text{else}\ e_2)$ is well typed, but if we want to push the application into the branches to make if behave a little more like goto (like the ANF translation does), this result is not well typed. We end up with the following failing type derivation.



This derivation needs to type check the application f applied to the branches e_1 and e_2. However, now that the application is pushed into the branches, we need to type check the application before the if statement unifies their types by replacing true and false by e. In essence, this transformation has forgotten that, by the time the expression $f\ e_1$ executes, the machine has evaluated e to true (and, analogously, e to false in the other branch).

Formalizing this intuition requires two extensions to type check dependent elimination: one for negatives types (Π and Σ, in ECC), and one for positive types (booleans, in ECC).

These translations were entirely in the source language, and thus in the source language font. They are not the ANF translation, merely source-to-source transformation that ought to be well typed. To support them, and ANF, we create a new **target language** in which these transformations, and the ANF translation, preserve typing.

For negatives types, it suffices to use *definitions* [Severi and Poll 1994], a standard extension to type theory that changes the typing rule for **let** to thread equalities into sub-derivations and resolve dependencies. The typing rule for **let** with definitions is:

$$\frac{\Gamma \vdash e : A \qquad \Gamma, \boxed{x = e} \vdash e' : A'}{\Gamma \vdash \text{let}\ x = e\ \text{in}\ e' : A'[x := e]}$$

The highlighted part, $x = e$, is the only difference from the standard dependent typing rule. This definition is introduced when type checking the body of the **let**, and can be used to solve type equivalence in sub-derivations, instead of only in the substitution $A'[x := e]$ in the "output" of the typing rule. While this is an extension to the type theory, it is a standard extension that is admissible in any Pure Type System (PTS) [Severi and Poll 1994], and is a feature already found in dependently typed languages such as Coq. The admissibility means it's uninteresting from a theoretical expressivity perspective, but it's important to a compiler IL where syntactic form is meaningful. With this addition, the transformation of f snd e type checks in the target language.

$$\frac{\dfrac{}{y = e \vdash \text{snd}\ y : B[x := \text{fst}\ e]} \qquad \dfrac{}{y = e, z = \text{snd}\ y \vdash f\ z : C}}{\dfrac{y = e \vdash \text{let}\ z = \text{snd}\ y\ \text{in}\ f\ z : C[z := \text{snd}\ y]}{\vdash \text{let}\ y = e\ \text{in let}\ z{=}\text{snd}\ y\ \text{in}\ f\ z : C[z{:=}\text{snd}\ y][y := e]}}$$

For positive types, we need to record an equality between the term being eliminated and its canonical form. For booleans, we need the following stronger typing rule for **if**.

$$\frac{\Gamma, x : \text{bool} \vdash B : \text{Type}_i \qquad \Gamma \vdash e : \text{bool} \qquad \Gamma, \boxed{e = \text{true}} \vdash e_1 : B[x := \text{true}] \qquad \Gamma, \boxed{e = \text{false}} \vdash e_2 : B[x := \text{false}]}{\Gamma \vdash \text{if}\ e\ \text{then}\ e_1\ \text{else}\ e_2 : B[x := e]}$$

The two highlighted portions of the rule are modifications over the standard typing rule. This rules introduces an equality or coercion from a term *te* that appears in the calling context's type to the canonical form known in the branches. This allows pushing the context into the branches. This rule essentially implements the convoy pattern [Chlipala 2013], but has a syntactic form suitable for compilation.

Unlike definitions, equalities require an elimination form that is inserted into transformed code. Equalities are eliminated by the term $\mathbf{subst}\ \mathbf{e}_1=\mathbf{e}_2\ \mathbf{e}$, which behaves like the function of the same name on the identity type. Returning to our earlier example `if` expression, with the modified typing rule for `if`, the following now type checks.

$$\frac{\dfrac{\mathbf{e}=\mathbf{true}\vdash \mathbf{subst}\ \mathbf{e}=\mathbf{true}\ \mathbf{e}_1 : \mathbf{B}[\mathbf{x}:=\mathbf{e}] \qquad \mathbf{f} : \mathbf{B}[\mathbf{x}:=\mathbf{e}]\to \mathbf{C}}{\mathbf{e}=\mathbf{true}\vdash \mathbf{f}\ \mathbf{subst}\ \mathbf{e}=\mathbf{true}\ \mathbf{e}_1 : \mathbf{C}[\mathbf{x}:=\mathbf{true}]}\ \ldots}{\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ (\mathbf{f}\ \mathbf{e}_1)\ \mathbf{else}\ (\mathbf{f}\ \mathbf{e}_2) : \mathbf{C}[\mathbf{x}:=\mathbf{e}]}$$

In related work, the encodings of machines steps are either missing, inconsistent, require additional axioms, or require restricting the translation or theories. Our encoding avoids all of these problems while supporting ANF. The general lesson is that dependent-type-preserving translations must carefully consider how to encode and record machine steps, *i.e.*, how to reflect a step of evaluation in the machine as an equality in the type system. In Section 7, we explain related work in terms of machine steps.

Formalizing Type Preservation for the ANF Translation. Formalizing the ANF type-preservation argument is tricky, since the translation is indexed by a (non-first-class) continuation, and essentially performs the translation "inside-out" by building up the translated term in the continuation. In the source, looking at an expression such as $\mathsf{snd}\,\mathbf{e}$, we do not know whether the expression is embedded in a larger context. This doesn't matter in the source, since we can compose by nesting expressions. But in ANF, we can no longer compose expressions by nesting and instead must compose expressions with **let**. To formalize the ANF translation, it helps to have a compositional syntax for translating an expression and composing it with an unknown context. Similarly, to reason about that translation, it helps to have a compositional way to reason about the types of an expression and the unknown context.

To make the translation compositional, we index the ANF translation by a target language (non-first-class) *continuation* $\mathbf{K}$ representing the rest of the computation in which a translated expression will be used.[5] A continuation $\mathbf{K}$ is a program with a hole (single linear variable) $[\cdot]$, and can be composed with a computation $\mathbf{N}$, written $\mathbf{K}[\mathbf{N}]$, to form a program $\mathbf{M}$. By keeping continuations non-first-class, we ensure that continuations must be used linearly and avoid control effects, which are known to cause inconsistency with dependent types [Barthe and Uustalu 2002; Herbelin 2005]. In ANF, there are only two continuations: either $[\cdot]$ or $\mathbf{let}\ \mathbf{x}=[\cdot]\ \mathbf{in}\ \mathbf{M}$. Using continuations, we define ANF translation for Σ types and second projections as follows.

$$[\![\Sigma\,\mathbf{x} : \mathbf{A}.\ \mathbf{B}]\!]\ \mathbf{K} = \mathbf{K}[\Sigma\,\mathbf{x} : [\![\mathbf{A}]\!].\ [\![\mathbf{B}]\!]]$$

$$[\![\mathsf{snd}\,\mathbf{e}]\!]\ \mathbf{K} = [\![\mathbf{e}]\!]\ \mathbf{let}\ \mathbf{y}=[\cdot]\ \mathbf{in}\ \mathbf{K}[\mathsf{snd}\,\mathbf{y}]$$

This allows us to focus on composing the primitive operations instead of reassociating **let** bindings.

For compositional reasoning, we develop a type system for continuations. The key typing rule is the following.

$$\frac{\Gamma\vdash \mathbf{N} : \mathbf{A} \qquad \Gamma, \mathbf{y}=\mathbf{M}'\vdash \mathbf{M} : \mathbf{B}}{\Gamma\vdash \mathbf{let}\ \mathbf{y}=[\cdot]\ \mathbf{in}\ \mathbf{M} : (\mathbf{M}' : \mathbf{A})\Rightarrow \mathbf{B}}\ [\textsc{K-Bind}]$$

The type $(\mathbf{M}' : \mathbf{A})\Rightarrow \mathbf{B}$ of continuations describes that the continuation must be composed with the term $\mathbf{M}'$ of type $\mathbf{A}$, and the result will be of type $\mathbf{B}$. Note that this type allows us to introduce the definition $\mathbf{y}=\mathbf{M}'$ via the type, before we know how that the continuation is used.[6] We discuss this

[5] This how ANF translation is implemented in Scheme by Flanagan et al. [1993], although their mathematical presentation is as a reduction system.

[6] This is essentially a singleton type, but we avoid explicit encoding with singleton types to focus on the intuition—machine steps—and avoid complicating the IL syntax.

Universes	U	::=	$\mathsf{Prop} \mid \mathsf{Type}_i$
Expressions	e, A, B	::=	$x \mid \mathsf{U} \mid \Pi\, x : A.\, B \mid \lambda\, x : A.\, e \mid e\ e \mid \Sigma\, x : A.\, B \mid \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B$
		|	$\mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{bool} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
Environments	Γ	::=	$\cdot \mid \Gamma, x : A \mid \Gamma, x = e$

Fig. 1. ECC Syntax

rule further in Section 4. Lemma 4.8 (Cut) expresses that continuation typing is not an extension to the target type theory, which is important to ensure that the ANF translation can be applied in practice.

The key lemma to prove type preservation is the following.

Lemma 2.1. *If* $\Gamma \vdash e : A$ *and* $\Gamma, \mathtt{defs}(\llbracket e \rrbracket) \vdash \mathbf{K} : (\mathtt{hole}(\llbracket e \rrbracket) : \llbracket A \rrbracket) \Rightarrow \mathbf{B}$*, then* $\Gamma \vdash \llbracket e \rrbracket\ \mathbf{K} : \mathbf{B}$.

This lemma captures the fact that each time we build a new $\mathbf{K}$ in the ANF translation, we must show it is well typed. Proving that $\mathbf{K}$ has the above type requires proving $\Gamma, \mathtt{defs}(\llbracket e \rrbracket) \vdash \mathtt{hole}(\llbracket e \rrbracket) : \llbracket A \rrbracket$ (where $\llbracket e \rrbracket$ is the ANF translation of e with the empty continuation). The auxiliary functions $\mathtt{hole}(\mathbf{M})$ and $\mathtt{defs}(\mathbf{M})$ capture the idea that an ANF term exports some definitions $\mathtt{defs}(\mathbf{M})$ into its body $\mathtt{hole}(\mathbf{M})$, and allow us to easily talk about these different aspects of an ANF term. For our running example, the essence of the type preservation argument means proving $\Gamma, \mathtt{defs}(\llbracket \mathsf{snd}\ e \rrbracket) \vdash \mathbf{snd\ y} : \llbracket B[x := \mathsf{fst}\ e] \rrbracket$. Recall that $\mathbf{snd\ y} : \llbracket B \rrbracket[\mathbf{x} := \mathbf{fst\ y}]$. Definitions allow us to prove that $\Gamma, \mathtt{defs}(\llbracket \mathsf{snd}\ e \rrbracket) \vdash \mathbf{y} \equiv \llbracket e \rrbracket$, which implies $\mathbf{fst\ y} \equiv \llbracket \mathsf{fst}\ e \rrbracket$. Therefore, combined with compositionality ($\llbracket B[x := \mathsf{fst}\ e] \rrbracket \equiv \llbracket B \rrbracket[\mathbf{x} := \llbracket \mathsf{fst}\ e \rrbracket]$, Lemma 5.4), the proof is complete!

3 SOURCE: ECC WITH DEFINITIONS

Our source language, ECC, is Luo's Extended Calculus of Constructions (ECC) [Luo 1990] extended with dependent elimination of booleans and with definitions [Severi and Poll 1994]. We typeset ECC in a non-bold, blue, sans-serif font. We present the syntax of ECC in Figure 1. ECC extends the Calculus of Constructions (CC) [Coquand and Huet 1988] with Σ types (strong dependent pairs) and an infinite predicative hierarchy of universes. There is no explicit phase distinction, *i.e.*, there is no syntactic distinction between *terms*, which represent run-time expressions, and *types*, which classify terms. However, we usually use the meta-variable e to evoke a term, and the meta-variables A and B to evoke a type. The language includes one impredicative universe, Prop, and an infinite hierarchy of predicative universes Type_i. The syntax of expressions e includes names x, universes U, dependent function types $\Pi\, x : A.\, B$, functions $\lambda\, x : A.\, e$, application $e_1\ e_2$, dependent pair types $\Sigma\, x : A.\, B$, dependent pairs $\langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B$, first $\mathsf{fst}\ e$ and second $\mathsf{snd}\ e$ projections of dependent pairs, dependent let $\mathsf{let}\ x = e\ \mathsf{in}\ e'$, the boolean type bool, the boolean value true and false, and dependent if $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. For brevity, we omit the type annotation on dependent pairs, as in $\langle e_1, e_2 \rangle$. Note that let-bound definitions do not include type annotations; this is not standard, but type checking is still decidable [Severi and Poll 1994], and it simplifies our ANF translation[7].

For simplicity, we assume uniqueness of names and ignore capture-avoiding substitution. This is standard practice, but is worth pointing out explicitly anyway.

In Figure 2, we give the reductions $\Gamma \vdash e \rhd e'$ for ECC, which are entirely standard. As usual, we extend reduction to conversion by defining $\Gamma \vdash e \rhd^* e'$ to be the reflexive, transitive, compatible closure of reduction $\rhd$. The conversion relation is used to compute equivalence between types, but we can also view it as the operational semantics for the language. We define $\mathsf{eval}(e)$ as the evaluation function for whole-programs using conversion, which we use in our compiler correctness proof.

In Figure 3 we define *definitional equivalence* (or just *equivalence*) $\Gamma \vdash e \equiv e'$ as conversion up to η-equivalence. We usually we the notation $e_1 \equiv e_2$ for equivalence, eliding the environment when

[7] We describe in Section 5 how to extend the ANF translation to support annotated `let`.

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

$$\begin{array}{rcll} x & \rhd_\delta & e & \text{where } x = e \in \Gamma \\ (\lambda\, x : A.\, e_1)\; e_2 & \rhd_\beta & e_1[x := e_2] & \\ \mathsf{fst}\ \langle e_1, e_2\rangle & \rhd_{\pi_1} & e_1 & \\ \mathsf{snd}\ \langle e_1, e_2\rangle & \rhd_{\pi_2} & e_2 & \\ \mathsf{let}\ x = e\ \mathsf{in}\ e' & \rhd_\zeta & e'[x := e] & \\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_1} & e_1 & \\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_2} & e_2 & \end{array}$$

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

$$\cdots \quad \frac{\Gamma, x = e \vdash e_1 \rhd^* e_2}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e_1 \rhd^* \mathsf{let}\ x = e\ \mathsf{in}\ e_2}\ [\textsc{Red-Cong-Let}]$$

$\boxed{\mathsf{eval}(e) = v}$

$$\mathsf{eval}(e) \quad = \quad v \quad \text{where } e \rhd^* v \text{ and } v \not\rhd v'$$

Fig. 2. ECC Reduction, Conversion, and Evaluation (excerpts)

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

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

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

$\boxed{\Gamma \vdash A \preceq B}$

$$\frac{\Gamma \vdash A \equiv B}{\Gamma \vdash A \preceq B}\ [\preceq\text{-}\equiv] \qquad \cdots \qquad \frac{}{\Gamma \vdash \mathsf{Type}_i \preceq \mathsf{Type}_{i+1}}\ [\preceq\textsc{-Cum}]$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \qquad \Gamma, x_1 : A_2 \vdash B_1 \preceq B_2[x_2 := x_1]}{\Gamma \vdash \Pi\, x_1 : A_1.\, B_1 \preceq \Pi\, x_2 : A_2.\, B_2}\ [\preceq\textsc{-Pi}]$$

Fig. 3. ECC Equivalence and Subtyping (excerpts)

it is obvious or unnecessary. We also define *cumulativity* (subtyping) $\Gamma \vdash A \preceq B$, to allow types in lower universes to inhabit higher universes.

We define the type system for ECC in Figure 4, which is mutually defined with well-formedness of environments in Figure 5. The typing rules are entirely standard for a dependent type system. Note that types themselves, such as $\Pi\, x : A.\, B$ have types (called universes), and universes also have types which are higher universes. In [Ax-Prop], the type of Prop is Type_0, and in [Ax-Type], the type of each universe Type_i is the next higher universe Type_{i+1}. Note that we have impredicative function types in Prop, given by [Prod-Prop]. For this work, we ignore the Set vs Prop distinction used in some type theories, such as Coq's, although adding it causes no difficulty. Note that the rules for application, [App], second projection, [Snd], let, [Let], and if [If] substitute sub-expressions into the type system. These are the key typing rules that introduce difficulty in type-preserving compilation for dependent types.

4 TARGET: ECC WITH ANF SUPPORT

Our target language, ECC^A, is a variant of ECC with a modified typing rule for dependent **if** that introduces equalities between terms (akin to the identity type), and an elimination form for assumed

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_0}\ [\textsc{Ax-Prop}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ [\textsc{Ax-Type}] \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ [\textsc{Var}]$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A, x = e \vdash e' : B}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : B[x := e]}\ [\textsc{Let}] \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x : A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi\, x : A.\, B : \mathsf{Prop}}\ [\textsc{Prod-Prop}]$$

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

$$\frac{\Gamma \vdash e : \Pi\, x : A'.\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\ e' : B[x := e']}\ [\textsc{App}] \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x : A \vdash B : \mathsf{Type}_i}{\Gamma \vdash \Sigma\, x : A.\, B : \mathsf{Type}_i}\ [\textsc{Sig}]$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[x := e_1]}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B : \Sigma\, x : A.\, B}\ [\textsc{Pair}] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{fst}\ e : A}\ [\textsc{Fst}] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\ e : B[x := \mathsf{fst}\ e]}\ [\textsc{Snd}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{bool} : \mathsf{Type}_0}\ [\textsc{Bool}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ [\textsc{True}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ [\textsc{False}]$$

$$\frac{\Gamma, x : \mathsf{bool} \vdash B : U \qquad \Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : B[x := \mathsf{true}] \qquad \Gamma \vdash e_2 : B[x := \mathsf{false}]}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : B[x := e]}\ [\textsc{If}]$$

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

Fig. 4. ECC Typing

$\boxed{\vdash \Gamma}$

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

Fig. 5. ECC Well-Formed Environments

Universes	U	::=	$\mathbf{Prop} \mid \mathbf{Type}_i$
Values	V	::=	$\mathbf{x} \mid \mathbf{U} \mid \lambda\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \Pi\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \Sigma\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \langle \mathbf{V}, \mathbf{V} \rangle \mid \mathbf{bool} \mid \mathbf{true} \mid \mathbf{false}$
Computations	N	::=	$\mathbf{V} \mid \mathbf{V}\ \mathbf{V} \mid \mathbf{fst}\ \mathbf{V} \mid \mathbf{snd}\ \mathbf{V} \mid \mathbf{subst}_{\mathbf{V}=\mathbf{V}}\ \mathbf{V}$
Configurations	M	::=	$\mathbf{N} \mid \mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{M} \mid \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2$
Continuations	K	::=	$[\cdot] \mid \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M}$
Environments	Γ	::=	$\cdot \mid \Gamma, \mathbf{x} : \mathbf{A} \mid \Gamma, \mathbf{x} = \mathbf{V} \mid \Gamma, \mathbf{V} = \mathbf{V}$

Fig. 6. ECC^A Syntax

equalities. These extensions are similar to the extensions used by Cong and Asai [2018a] to support CPS translation with dependent pattern matching, which we discuss further in Section 7. While ECC^A supports ANF syntax, the full language is not ANF restricted; it has the same syntax as ECC, and uses the usual definitional equivalence and conversion relation for type checking. We do not restrict the full language because maintaining ANF while type checking adds needless complexity; instead, we show that our compiler generates only ANF restricted terms in ECC^A, and define a separate ANF-preserving machine-like semantics for evaluating programs in ANF.

$$\ldots \quad \frac{\Gamma, \mathbf{x} : \mathbf{bool} \vdash \mathbf{B} : \mathbf{U} \qquad \Gamma \vdash \mathbf{e} : \mathbf{bool} \qquad \Gamma, \mathbf{e} = \mathbf{true} \vdash \mathbf{e}_1 : \mathbf{B}[\mathbf{x} := \mathbf{true}] \qquad \Gamma, \mathbf{e} = \mathbf{false} \vdash \mathbf{e}_2 : \mathbf{B}[\mathbf{x} := \mathbf{false}]}{\Gamma \vdash \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2 : \mathbf{B}[\mathbf{x} := \mathbf{e}]}\ [\textsc{If}]$$

$$\frac{\Gamma, \mathbf{x} : \mathbf{A} \vdash \mathbf{B} : \mathbf{U} \qquad \Gamma \vdash \mathbf{e}_1 : \mathbf{A} \qquad \Gamma \vdash \mathbf{e}_2 : \mathbf{A} \qquad \Gamma \vdash \mathbf{e} : \mathbf{B}[\mathbf{x} := \mathbf{e}_1] \qquad \mathbf{e}_1 = \mathbf{e}_2 \in \Gamma}{\Gamma \vdash \mathbf{subst}_{\mathbf{e}_1 = \mathbf{e}_2}\ \mathbf{e} : \mathbf{B}[\mathbf{x} := \mathbf{e}_2]}\ [\textsc{Subst}]$$

Fig. 7. ECC^A Typing (excerpts)

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

$$\ldots \quad \frac{\Gamma \vdash \mathbf{e} \equiv \mathbf{e}'}{\Gamma \vdash \mathbf{subst}_{\mathbf{e}_1 = \mathbf{e}_2}\ \mathbf{e} \equiv \mathbf{e}'}\ [\equiv\text{-}\textsc{Subst}_1] \qquad \frac{\mathbf{true} = \mathbf{false} \in \Gamma}{\Gamma \vdash \mathbf{e}_1 \equiv \mathbf{e}_2}\ [\equiv\text{-}\textsc{Absurd}_1]$$

Fig. 8. ECC^A Equivalence (excerpts)

We can imagine the compilation process as either: (1) generating ANF syntax in ECC^A from ECC, or (2) as first embedding ECC in ECC^A and then rewriting ECC^A terms into ANF. In Section 5 we present the compiler as process (1), a compiler from ECC to ANF ECC^A. In this section we develop most of the supporting meta-theory necessary for ANF as intra-language equivalences and process (2) may be a more helpful intuition. We typeset ECC^A in a **bold, red, serif font**; in later sections, we reserve this font exclusively for the ANF restricted ECC^A.

Note that because the whole language is not restricted to ANF, definitional equivalence is not suitable for equational reasoning about run-time terms (*e.g.*, reasoning about optimizations), unless we ANF translate any terms rewritten by definitional equivalence. This ability to break ANF locally to support reasoning is similar to the language F_J of Maurer et al. [2017], which does not enforce ANF syntactically, but supports ANF transformation and optimization with join points.

We give the ANF syntax for ECC^A in Figure 6. We impose a syntactic distinction between *values* $\mathbf{V}$ which do not reduce, *computations* $\mathbf{N}$ which eliminate values and can be composed using *continuations* $\mathbf{K}$, and *configurations* $\mathbf{M}$ which represent the machine configurations executed by the ANF machine semantics. A continuation $\mathbf{K}$ is a program with a hole, and is composed $\mathbf{K}[\mathbf{N}]$ with a computation $\mathbf{N}$ to form a configuration $\mathbf{M}$. For example, $(\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x})[\mathbf{N}] = (\mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x})$. Since continuations are not first-class objects, we cannot express control effects—continuations are syntactically guaranteed to be used linearly. Note that despite the *syntactic* distinctions, we still do not enforce a *phase* distinction—configurations (programs) can appear in types.

We give the new typing rules in Figure 7. The key change in ECC^A is in the typing rule for dependent if. The typing rule for $\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2$ introduces an unnamed equality in each branch. This records the information that in $\mathbf{e}_1$ the target $\mathbf{e}$ being eliminated will be equal to $\mathbf{true}$ ($\mathbf{e} = \mathbf{true}$), and in $\mathbf{e}_2$ we know statically that $\mathbf{e} = \mathbf{false}$. These record a machine step: the machine will have reduced $\mathbf{e}$ to $\mathbf{true}$ before jumping to the first branch, and reduced $\mathbf{e}$ to $\mathbf{false}$ before jumping to the second branch. This is necessary to support the ANF transformation of dependent if.

The typing rule for $\mathbf{subst}$ is essentially a standard eliminator of the identity type. The typing rule for an expression $\mathbf{subst}_{\mathbf{e}_1=\mathbf{e}_2}\ \mathbf{e}$ replaces $\mathbf{e}_1$ by $\mathbf{e}_2$ in the type of $\mathbf{e}$, if we have assumed $\mathbf{e}_1 = \mathbf{e}_2$. To ensure normalization, $\mathbf{subst}_{\mathbf{e}_1=\mathbf{e}_2}\ \mathbf{e}$ can only reduce when $\mathbf{e}_1$ and $\mathbf{e}_2$ are syntactically identical, analogous to requiring the canonical proof $\mathbf{refl}$ of the identity type. We use unnamed equality assumptions instead of the identity type to simplify the syntax of dependent if.

We require several additional equivalence rules; the two key rules are shown in Figure 8. These rules are necessary for proving ANF is type preserving in the presence of dependent if. The rule $[\equiv\text{-}\textsc{Subst}_1]$ (and a symmetric rule $[\equiv\text{-}\textsc{Subst}_2]$) allows reasoning about equivalence of $\mathbf{subst}$ without

$\boxed{\mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle = \mathbf{M}}$

$$\begin{aligned}
\mathbf{K}\langle\!\langle \mathbf{N} \rangle\!\rangle &\stackrel{\text{def}}{=} \mathbf{K}[\mathbf{N}] \\
\mathbf{K}\langle\!\langle \mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{M} \rangle\!\rangle &\stackrel{\text{def}}{=} \mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle \\
\mathbf{K}\langle\!\langle \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 \rangle\!\rangle &\stackrel{\text{def}}{=} \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ (\mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{true}}\ \mathbf{x}][\mathbf{M}_1 // \mathbf{x}])\ \mathbf{else}\ (\mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{false}}\ \mathbf{x}][\mathbf{M}_2 // \mathbf{x}])
\end{aligned}$$

$\boxed{\mathbf{K}\langle\!\langle \mathbf{K} \rangle\!\rangle = \mathbf{K}}$

$$\begin{aligned}
\mathbf{K}\langle\!\langle [\cdot] \rangle\!\rangle &\stackrel{\text{def}}{=} \mathbf{K} \\
\mathbf{K}\langle\!\langle \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M} \rangle\!\rangle &\stackrel{\text{def}}{=} \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle
\end{aligned}$$

$\boxed{\mathbf{M}[\mathbf{M}//\mathbf{x}] = \mathbf{M}}$

$$\mathbf{M}[\mathbf{M}'//\mathbf{x}] \stackrel{\text{def}}{=} (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M})\langle\!\langle \mathbf{M}' \rangle\!\rangle$$

Fig. 9. Composition of Configurations

reducing it, maintaining normalization in the presence of possibly inconsistent assumed equalities. The second rule is required in exactly two places in the type-preservation proof, and allows us to conclude any two terms are equivalent if we assume $\mathbf{true} = \mathbf{false}$ (or, symmetrically, $\mathbf{false} = \mathbf{true}$).[8]

In ANF, all continuations are left associated, so substitution can break ANF. Note that β-reduction takes an ANF configuration $\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\ \mathbf{M})\ \mathbf{V}]$ but would naïvely produce $\mathbf{K}[\mathbf{M}[\mathbf{x} := \mathbf{V}]]$. Substituting the term $\mathbf{M}[\mathbf{x} := \mathbf{V}]$, a *configuration*, into the continuation $\mathbf{K}$ could result in the non-ANF term $\mathbf{let}\ \mathbf{x} = \mathbf{M}\ \mathbf{in}\ \mathbf{M}'$. In ANF, configurations cannot be nested.

To ensure reduction preserves ANF, we define composition of a continuation $\mathbf{K}$ and a configuration $\mathbf{M}$, Figure 9, typically called *renormalization* in the literature [Kennedy 2007; Sabry and Wadler 1997]. When composing a continuation with a configuration, $\mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle$, we essentially unnest all continuations so they remain left associated. [9] When composing a continuation with an **if** statement, notice that the continuation is duplicated in the branches. This is the usual, naïve, presentation of ANF; we show later that the join-point optimization, which avoids this duplication, is admissible in ECC^A. Note that these definitions are simplified by our uniqueness-of-names assumption.

Digression on composition in ANF. In the literature, the composition operation $\mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle$ is usually introduced as *renormalization*, as if the only intuition for why it exists is "well, it happens that ANF is not preserved under β-reduction". It is not mere coincidence; the intuition for this operation is composition, and having a syntax for composing terms is not only useful for stating β-reduction, but useful for all reasoning about ANF! This should not come as a surprise—compositional reasoning is useful. The only surprise is that the composition operation is not the usual one used in programming language semantics, *i.e.*, substitution. In ANF, as in monadic normal form, substitution can be used to compose any expression with a *value*, since names are values and values can always be replaced by values. But substitution cannot just replace a name, which is a *value*, with a *computation* or *configuration*. That wouldn't be well-typed. So how do we compose computations with configurations? We can use **let**, as in $\mathbf{let}\ \mathbf{y} = \mathbf{N}\ \mathbf{in}\ \mathbf{M}$, which we can imagine as an explicit substitution. In monadic form, there is no distinction between computations and configurations, so the same term works to compose configurations. But in ANF, we have no object-level term to compose *configurations* or *continuations*. We cannot substitute a configuration $\mathbf{M}$ into a continuation $\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathbf{M}'$, since this would result in the non-ANF (but valid monadic) expression $\mathbf{let}\ \mathbf{y} = \mathbf{M}\ \mathbf{in}\ \mathbf{M}'$.

[8] As a personal remark, one author tried everything in their power to avoid this rule, but failed. Several alternatives, such as the standard **False** elimination typing rule, were considered, and would work, but this seems to be the most tasteful for a compiler IR, in our humble opinion.

[9] Some work uses an append notation, *e.g.*, $\mathbf{M} :: \mathbf{K}$ [Sabry and Wadler 1997], suggesting we are appending $\mathbf{K}$ onto the stack for $\mathbf{M}$; we prefer notation that evokes composition.

$\boxed{M \mapsto M'}$

$$\begin{array}{rcl} K[(\lambda\, x : A.\, M)\ V] & \mapsto_\beta & K\langle\!\langle M[x := V]\rangle\!\rangle \\ K[\mathsf{fst}\ \langle V_1, V_2\rangle] & \mapsto_{\pi_1} & K[V_1] \\ K[\mathsf{snd}\ \langle V_1, V_2\rangle] & \mapsto_{\pi_2} & K[V_2] \\ \mathsf{let}\ x = V\ \mathsf{in}\ M & \mapsto_\zeta & M[x := V] \\ \mathsf{if\ true\ then}\ M_1\ \mathsf{else}\ M_2 & \mapsto_{\iota_1} & M_1 \\ \mathsf{if\ false\ then}\ M_1\ \mathsf{else}\ M_2 & \mapsto_{\iota_2} & M_2 \\ K[\mathsf{subst}_{V=V}\ V'] & \mapsto_= & K[V'] \end{array}$$

$\boxed{M \mapsto^* M'}$

$$\frac{}{M \mapsto^* M}\ [\textsc{Red-Refl}] \qquad \frac{M \mapsto M_1 \qquad M_1 \mapsto^* M'}{M \mapsto^* M'}\ [\textsc{Red-Trans}]$$

$\boxed{\mathsf{eval}(M) = V}$

$$\mathsf{eval}(M) \quad = \quad V \quad \text{where } M \mapsto^* V \text{ and } V \not\mapsto V'$$

Fig. 10. ECC^A Evaluation

$\boxed{\Gamma \vdash K : (M : A) \Rightarrow B}$

$$\frac{}{\Gamma \vdash [\cdot] : (M : A) \Rightarrow A}\ [\textsc{K-Empty}] \qquad \frac{\Gamma \vdash M' : A \qquad \Gamma, y = M' \vdash M : B}{\Gamma \vdash \mathsf{let}\ y = [\cdot]\ \mathsf{in}\ M : (M' : A) \Rightarrow B}\ [\textsc{K-Bind}]$$

Fig. 11. ECC^A Continuation Typing

Instead, ANF requires a new operation to compose configurations: $K\langle\!\langle M\rangle\!\rangle$. This operation is more generally known as *hereditary substitution* [Watkins et al. 2003], a form of substitution that maintains canonical forms. So we can think of it as a form of substitution, or, simply, as composition.

In Figure 10, we present the call-by-value (CBV) evaluation semantics for ANF ECC^A terms. It is essentially standard, but recall that β-reduction produces a configuration M which must be composed with the existing continuation K. This semantics is for the run-time evaluation of configurations; during type checking, we continue to use the type system and conversion relation defined in Section 3.

4.1 Dependent Continuation Typing

The ANF translation manipulates continuations K as independent entities. To reason about them, and thus to reason about the translation, we introduce continuation typing, defined in Figure 11. The type $(M : A) \Rightarrow B$ of a continuation expresses that this continuation expects to be composed with a term equal to the configuration M of type A and returns a result of type B when completed. Normally, M is equivalent to some computation N, but it must be generalized to a configuration M to support dependent **if** expressions. This type formally expresses the idea that M is depended upon (in the sense introduced in Section 2) in the rest of the computation. For the empty continuation $[\cdot]$, M is arbitrary since an empty continuation has no "rest of the program" that could depend on anything.

Intuitively, what we want from continuation typing is a compositionality property—that we can reason about the types of configurations $K[N]$ (generally, for configurations $K\langle\!\langle M\rangle\!\rangle$) by composing the typing derivations for K and N. To get this property, a continuation type must express not merely the *type* of its hole A, but *which term* N will be bound in the hole. We see this formally from the typing rule [Let] (the same for ECC^A as for ECC in Section 3), since showing that $\mathsf{let}\ y = N\ \mathsf{in}\ M$ is well-typed requires showing that $y = N \vdash M$, that is, requires knowing the definition $y = N$. If we

$\boxed{[\![\mathbf{e}]\!]_M = e \text{ where } \Gamma \vdash \mathbf{e} : \mathbf{A}}$

$$[\![\mathbf{subst}\ \mathbf{e}_1{=}\mathbf{e}_2\ \mathbf{e}]\!]_M \stackrel{\text{def}}{=} \text{subst}\ p\ [\![\mathbf{e}]\!]_M \quad \text{where } p : [\![\mathbf{e}_1]\!]_M = [\![\mathbf{e}_2]\!]_M \in [\![\Gamma]\!]_M$$

$$[\![\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2]\!]_M \stackrel{\text{def}}{=} (\text{if}\ [\![\mathbf{e}]\!]_M\ \text{then}\ (\lambda p : \text{true} = [\![\mathbf{e}]\!]_M.\ \text{subst}\ (\text{if-eta1}\ [\![\mathbf{e}_1]\!]_M\ [\![\mathbf{e}_2]\!]_M\ p)\ [\![\mathbf{e}_1]\!]_M)\ \text{else}\ (\lambda p : \text{false} = [\![\mathbf{e}]\!]_M.\ \text{subst}\ (\text{if-eta2}\ [\![\mathbf{e}_1]\!]_M\ [\![\mathbf{e}_2]\!]_M\ p)\ [\![\mathbf{e}_2]\!]_M))\ \text{refl}\ [\![\mathbf{e}]\!]_M$$

Auxiliary CIC definitions

$$\cdots \qquad \frac{\Gamma \vdash p : e_1 = e_2 \qquad \Gamma \vdash e : B[x := e_1]}{\Gamma \vdash \text{subst}\ p\ e : B[x := e_2]}\ [\text{DEF-SUBST}]$$

$$\frac{\Gamma, x : bool \vdash B : U \qquad \Gamma \vdash e_1 : B[x := \text{true}] \qquad \Gamma \vdash e_2 : B[x := \text{false}] \qquad \Gamma \vdash p : \text{true} = e}{\Gamma \vdash \text{if-eta1}\ e_1\ e_2\ p : \text{subst}\ p\ e_1 = \text{if}\ e\ \text{then}\ e_1\ \text{else}\ e_2}\ [\text{DEF-IF-ETA1}]$$

$$\frac{\Gamma, x : bool \vdash B : U \qquad \Gamma \vdash e_1 : B[x := \text{true}] \qquad \Gamma \vdash e_2 : B[x := \text{false}] \qquad \Gamma \vdash p : \text{false} = e}{\Gamma \vdash \text{if-eta2}\ e_1\ e_2\ p : \text{subst}\ p\ e_2 = \text{if}\ e\ \text{then}\ e_1\ \text{else}\ e_2}\ [\text{DEF-IF-ETA2}]$$

Fig. 12. ECC^A Model in CIC (excerpts)

omit the expression N from the type of a continuation, we know there are some configurations K[N] that we cannot type check *compositionally*. Intuitively, if all we knew about y was its type, we would be in exactly the situation of trying to type check a continuation that has abstracted some dependent type that depends on the *specific* N into one that depends on an *arbitrary* y. We prove that our continuation typing is compositional in this way, Lemma 4.8 (Cut).

Note that the result of a continuation type cannot depend on the term that will be plugged in for the hole, *i.e.*, for a continuation K : (N : A) ⇒ B, B does not depend on N. To see this, first note that the initial continuation must be empty and thus *cannot* have a result type that depends on its hole. The ANF translation will take this initial empty continuation and compose it with intermediate continuations K′. Since composing any continuation K : (N : A) ⇒ B with any continuation K′ results in a new continuation with the final result type B, then the composition of any two continuations cannot depend on the type of the hole. This is similar to how, in CPS, the answer type doesn't matter and might as well be ⊥.

4.2 Meta-Theory

4.2.1 Consistency. To demonstrate that the new typing rule for dependent if is consistent, we give a syntactic model of ECC^A in CIC, and formalize the proofs of key properties of the model in Coq. The model essentially implements the new dependent if using the *convoy pattern* [Chlipala 2013], and implements assumed equalities as the identity type. The model in Coq uses propositional equivalence, so constructing a syntactic model relies on equivalence reflection to translate these into the required definitional equivalences. ECC^A itself does not explicitly rely on equivalence reflection, and it is not obvious whether a model exists that does not rely on equivalence reflection.

The essence of the model is given in Figure 12. There are only two interesting rules. The form $\mathbf{subst}\ \mathbf{e}_1{=}\mathbf{e}_2\ \mathbf{e}$ is simply translated into a call to the function subst, implemented in CIC, applied to some variable p of the identity type that is in scope in the model. Each if expression $\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2$ is implemented using the convoy pattern, transforming each if expression into a new if expression that returns a function expecting a proof that $[\![\mathbf{e}]\!]_M$ is equal to true in the first branch and false in the second branch. The if expression is then immediately applied to refl. The model relies on auxiliary

definitions in CIC, including subst, if-eta1 and if-eta2, whose types are given as inference rules in Figure 12, and whose full Coq implementations are in [10]. Note that the model for ECC^A's **if** is not valid ANF, so it does not suffice to merely *use* the convoy pattern if we want to take advantage of ANF for compilation.

We show this is a syntactic model using the usual recipe, which is explained well by Boulier et al. [2017]: we show the translation from ECC^A to CIC preserves equivalence, typing, and the definition of False (the empty type). This means that if ECC^A were inconsistent, then we could translate the proof of **False** into a proof of False in CIC, but no such proof exists in CIC, so ECC^A is consistent.

We use the usual definition of **False** as **Π x : Prop . x**, and the same in CIC. It is trivial that the definition is preserved.

Lemma 4.1 (Model Preserves Falseness). $[\![\mathbf{False}]\!]_M \equiv \mathit{False}$

The essence of showing both that equivalence is preserved and that typing is preserved is in showing that the auxiliary definitions in Figure 12 exist and are well typed. We give these definitions in Coq in [11], and the equivalences hold and are well typed without any additional axioms. Note, however, that Lemma 4.2 is stated in terms of definitional equivalence, while the Coq implementation uses propositional equivalence. This means interpreting our Coq implementation as a model requires equivalence reflection.

Lemma 4.2 (Model Preserves Equivalence). *If* $\mathbf{e}_1 \equiv \mathbf{e}_2$ *then* $[\![\mathbf{e}_1]\!]_M \equiv [\![\mathbf{e}_2]\!]_M$.

Lemma 4.3 (Model Preserves Typing). *If* $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ *then* $[\![\mathbf{\Gamma}]\!]_M \vdash [\![\mathbf{e}]\!]_M : [\![\mathbf{A}]\!]_M$.

Theorem 4.4 (Consistency). *There is no* **e** *such that* $\cdot \vdash \mathbf{e} : \mathbf{False}$

4.2.2 Correctness of ANF Evaluation. In ECC^A, we have an ANF evaluation semantics for run time and a separate definitional equivalence and reduction system for type checking. In this section, we prove that these two coincide: running in our ANF evaluation semantics produces a value definitionally equivalent to the original term.

When computing definitional equivalence, we end up with terms that are not in ANF, and can no longer be used in the ANF evaluation semantics. This is not a problem—we could always ANF translate the resulting term if needed—but can be confusing when reading equations. To make it clear which terms are in ANF and which are not, we leave terms and subterms that are in ANF in the **target language font**, and write terms or subterms that are not in ANF in the source language font. Meta-operations like substitution may be applied to ANF (**red**) terms, but result in non-ANF (blue) terms. Since substitution leaves no visual trace of its blueness, we wrap such terms in a distinctive language boundary such as $\mathcal{ST}(\mathbf{M}[\mathbf{x} := \mathbf{M'}])$ and $\mathcal{ST}(\mathbf{K}[\mathbf{M}])$. The boundary indicates the term is a target (ANF) ($\mathcal{T}$) term on the inside but a source (non-ANF) ($\mathcal{S}$) term on the outside. The boundary is only meant to communicate with the reader that a term is no longer in ANF; formally, $\mathcal{ST}(\mathbf{e}) = \mathbf{e}$.

The heart of the correctness proof is actually *naturality*, a property found in the literature on continuations and CPS that essentially expresses freedom from control effects (*e.g.*, Thielecke [2003] explain this well). This seems to be related to linearity and thunkability in the call-by-push-value literature; a recent draft by Pédrot and Tabareau [2017] explains how these properties relate to CPS translation in dependent type theory.

Lemma 4.5 is the formal statement of naturality in ANF: composing a term **M** with its continuation **K** in ANF is equivalent to running **M** to a value and substituting the result into the continuation

[10] https://www.williamjbowman.com/downloads/anf-sigma-techrpt.pdf
[11] https://www.williamjbowman.com/downloads/anf-sigma-techrpt.pdf

K. Formally, this states that composing continuations in ANF is sound with respect to standard substitution.

Lemma 4.5 (Naturality). $\mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle \equiv \mathcal{ST}(\mathbf{K}[\mathbf{M}])$

Proof. By induction on the structure of M

Case: $\mathbf{M} = \mathbf{N}$ trivial

Case: $\mathbf{M} = \mathbf{let\ x = N'\ in\ M'}$

Must show that $\mathbf{let\ x = N'\ in\ K}\langle\!\langle \mathbf{M'} \rangle\!\rangle \equiv \mathcal{ST}(\mathbf{K}[\mathbf{let\ x = N'\ in\ M}])$. This requires breaking ANF while computing equivalence.

$$
\begin{aligned}
& \mathbf{let\ x = N'\ in\ K}\langle\!\langle \mathbf{M'} \rangle\!\rangle \\
\rhd_\zeta\ & \mathcal{ST}(\mathbf{K}\langle\!\langle \mathbf{M'} \rangle\!\rangle[\mathbf{x := N'}]) && \text{note: this substitution is undefined in ANF} && (1)\\
= \ & \mathbf{K}\langle\!\langle \mathcal{ST}(\mathbf{M'}[\mathbf{x := N'}]) \rangle\!\rangle && \text{by uniqueness of names} && (2)\\
\lhd^*\ & \mathcal{ST}(\mathbf{K}[\mathbf{let\ x = N'\ in\ M}]) && \text{by } \zeta\text{-reduction and congruence} && (3)
\end{aligned}
$$

□

Next we show that our ANF evaluation semantics are sound with respect to definitional equivalence. This is also central to our later proof of compiler correctness. To do that, we first show that the small-step semantics are sound. Then we show soundness of the evaluation function.

Lemma 4.6 (Small-step soundness). *If* $\mathbf{M} \mapsto \mathbf{M'}$ *then* $\mathbf{M} \equiv \mathbf{M'}$

Proof. By cases on $\mathbf{M} \mapsto \mathbf{M'}$. Most cases follow easily from the ECC reduction relation and congruence. We give representative cases.

Case: $\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M}_1)\ \mathbf{V}] \mapsto_\beta \mathbf{K}\langle\!\langle \mathbf{M}_1[\mathbf{x := V}] \rangle\!\rangle$

Must show that $\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M}_1)\ \mathbf{V}] \equiv \mathbf{K}\langle\!\langle \mathbf{M}_1[\mathbf{x := V}] \rangle\!\rangle$

$$
\begin{aligned}
& \mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M}_1)\ \mathbf{V}] \\
\rhd^*\ & \mathcal{ST}(\mathbf{K}[\mathbf{M}_1[\mathbf{x := V}]]) && \text{by } \beta \text{ and congruence} && (4)\\
\equiv\ & \mathbf{K}\langle\!\langle \mathbf{M}_1[\mathbf{x := V}] \rangle\!\rangle && \text{by Lemma 4.5} && (5)
\end{aligned}
$$

Case: $\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] \mapsto_{\pi_1} \mathbf{K}[\mathbf{V}_1]$

Must show that $\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] \equiv \mathbf{K}[\mathbf{V}_1]$, which follows by $\rhd_{\pi_1}$ and congruence. □

Theorem 4.7 (Evaluation soundness). $\vdash \mathbf{eval(M)} \equiv \mathbf{M}$

Proof. By induction on the length n of the reduction sequence given by eval(M). Note that, unlike conversion, the ANF evaluation semantics have no congruence rules.

Case: $n = 0$ By [Red-Refl] and [≡].

Case: $n = i + 1$ Follows by Lemma 4.6 and the induction hypothesis. □

4.2.3 Admissibility of Continuation Typing. To prove that continuation typing is not an extension to the type system—*i.e.*, is admissible—we prove Lemma 4.8 and Lemma 4.12, that plugging a well-typed computation or configuration into a well-typed continuation results in a well-typed term of the expected type.

We first show Lemma 4.8 (Cut), which is simple. This lemma corresponds to the [Cut] rule (*e.g.*, found in sequent calculi), and tells us that our continuation typing allows for compositional reasoning about configurations K[N] whose result types do not depend on N. We generalize this lemma to configurations $\mathbf{K}\langle\!\langle \mathbf{N} \rangle\!\rangle$ shortly, but require more meta-theory to do so. The proof for this lemma is simple by definition.

Lemma 4.8 (Cut). *If* $\Gamma \vdash \mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ *and* $\Gamma \vdash \mathbf{N} : \mathbf{A}$ *then* $\Gamma \vdash \mathbf{K}[\mathbf{N}] : \mathbf{B}$.

$\boxed{\texttt{defs}(M) = \Gamma}$

$$\begin{aligned}
\texttt{defs}(N) &\stackrel{\text{def}}{=} \cdot \\
\texttt{defs}(\textbf{let}\ x = N\ \textbf{in}\ M) &\stackrel{\text{def}}{=} x = N, \texttt{defs}(M) \\
\texttt{defs}(\textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2) &\stackrel{\text{def}}{=} \cdot
\end{aligned}$$

$\boxed{\texttt{hole}(M) = M}$

$$\begin{aligned}
\texttt{hole}(N) &\stackrel{\text{def}}{=} N \\
\texttt{hole}(\textbf{let}\ x = N\ \textbf{in}\ M) &\stackrel{\text{def}}{=} \texttt{hole}(M) \\
\texttt{hole}(\textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2) &\stackrel{\text{def}}{=} \textbf{if}\ V\ \textbf{then}\ \texttt{hole}(M_1)\ \textbf{else}\ \texttt{hole}(M_2)
\end{aligned}$$

Fig. 13. Continuation Exports

Continuation typing seems to require that we compose a continuation $K : (N : A) \Rightarrow B$ *syntactically* with N, but we will need to compose with some $N' \equiv N$. It's preferably to prove this as a lemma instead of building it into continuation typing to get a nicer induction property for continuation typing. The proof is essentially that substitution respects equivalence.

Lemma 4.9 (Cut Modulo Equivalence). *If* $\Gamma \vdash K : (N : A) \Rightarrow B$, $\Gamma \vdash N : A$, $\Gamma \vdash N' : A$, *and* $\Gamma \vdash N \equiv N'$, *then* $\Gamma \vdash K[N'] : B$.

To reason inductively about ANF terms, we need to separate a configuration M into its exported definitions $\texttt{defs}(M)$ and its underlying computation $\texttt{hole}(M)$, which we define formally in Figure 13. The exported definitions represent all the machine steps (**let**-bound computations) that will happen "before" (in CBV, anyway) executing the body of M, while the $\texttt{hole}(M)$ is the body, the inner-most computation whose result will be bound when M is composed with another continuation. We define $\texttt{defs}(M)$ to be the sequence of definitions bound in the ANF term M. These are the definitions that will be in scope for a continuation K when composed with M, *i.e.*, in scope for K in $K\langle\!\langle M \rangle\!\rangle$. Note that $\texttt{hole}(M)$ will only be well typed in the environment for M extended with the definitions $\texttt{defs}(M)$.

We show that a configuration is nothing more than its exported definitions and underlying computation, *i.e.*, that in a context with the exports of $\texttt{defs}(M)$, $\texttt{hole}(M) \equiv M$. In essence, this lemma shows how ANF converts a dependency on a *configuration* M into a series of dependencies on *values*, *i.e.*, the names $x_0, \ldots, x_{n+1}$ in $\texttt{defs}(M)$. Note that the ANF guarantees that all dependent typing rules, like $V\ V' : B[x := V']$, only depend on values. This lemma allows us to recover the dependency on a configuration.

Lemma 4.10. *If* $\Gamma \vdash M : A$ *then* $\Gamma, \texttt{defs}(M) \vdash M \rhd^* \texttt{hole}(M)$

Proof. By induction on the syntax of M.

Case: N
Trivial, since $\texttt{hole}(N) = N$,

Case: $\textbf{let}\ x = N\ \textbf{in}\ M'$
Must show that $\Gamma, x = N, \texttt{defs}(M') \vdash \textbf{let}\ x = N\ \textbf{in}\ M' \rhd^* \texttt{hole}(M')$.
By [Red-Cong-Let], it suffices to show $\Gamma, x = N, \texttt{defs}(M'), x = N \vdash M' \rhd^* \texttt{hole}(M')$; recall that [Red-Cong-Let] introduces a redundant definition $x = N$.
Since names are unique, the second $x = N$ is redundant, and $\Gamma, x = N, \texttt{defs}(M') \vdash M' \rhd^* \texttt{hole}(M')$ follows by the induction hypothesis.

Case: $\textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2$
Must show that $\Gamma \vdash \textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \rhd^* \textbf{if}\ V\ \textbf{then}\ \texttt{hole}(M_1)\ \textbf{else}\ \texttt{hole}(M_2)$.
which follows by the congruence rule for if and the induction hypothesis. □

Corollary 4.11. *If* $\Gamma \vdash M : A$ *then* $\Gamma, \texttt{defs}(M) \vdash M \equiv \texttt{hole}(M)$.

The final lemma about continuation typing is also the key to why ANF is type preserving for dependent if. The heterogeneous composition operations necessarily performs the ANF translation on **if** expressions. The following lemma simply states that if a configuration $\mathbf{M}$ (and its decomposition into computation $\texttt{hole}(\mathbf{M})$ and definitions $\texttt{defs}(\mathbf{M})$) are well typed, and a continuation $\mathbf{K}$ is well typed at a compatible type, then the composition $\mathbf{K}\langle\!\langle\mathbf{M}\rangle\!\rangle$ is well typed. The proof is simple, except for the **if** case, which essentially must prove that ANF is type preserving for **if**.

Lemma 4.12 (Heterogeneous Cut). *If* $\Gamma \vdash \mathbf{M} : \mathbf{B}'$ *and* $\Gamma, \texttt{defs}(\mathbf{M}) \vdash \texttt{hole}(\mathbf{M}) : \mathbf{B}$ *and* $\Gamma, \texttt{defs}(\mathbf{M}) \vdash \mathbf{K} : (\texttt{hole}(\mathbf{M}) : \mathbf{B}) \Rightarrow \mathbf{C}$ *then* $\Gamma \vdash \mathbf{K}\langle\!\langle\mathbf{M}\rangle\!\rangle : \mathbf{C}$.

Proof. By induction on $\Gamma \vdash \mathbf{M} : \mathbf{B}'$

Case: $\Gamma \vdash \mathbf{N} : \mathbf{M}$, by Lemma 4.8.

Case: $\Gamma \vdash \mathbf{let}\ \mathbf{x}' = \mathbf{N}'\ \mathbf{in}\ \mathbf{M}' : \mathbf{B}''[\mathbf{x}' := \mathbf{N}']$

By the premise, we know:

(a) $\Gamma, \mathbf{x}' = \mathbf{N}' \vdash \mathbf{M}' : \mathbf{B}''$ by inversion

(b) $\Gamma, \mathbf{x}' = \mathbf{N}', \texttt{defs}(\mathbf{M}') \vdash \texttt{hole}(\mathbf{M}') : \mathbf{B}$ by definition

(c) $\Gamma, \mathbf{x}' = \mathbf{N}' \vdash \mathbf{K} : (\texttt{hole}(\mathbf{M}') : \mathbf{B}) \Rightarrow \mathbf{C}$ by definition

We must show that $\Gamma \vdash \mathbf{let}\ \mathbf{x}' = \mathbf{N}'\ \mathbf{in}\ \mathbf{K}\langle\!\langle\mathbf{M}'\rangle\!\rangle : \mathbf{B}''[\mathbf{x}' := \mathbf{N}']$.

By [Let], it suffices to show $\Gamma, \mathbf{x}' = \mathbf{N}' \vdash \mathbf{K}\langle\!\langle\mathbf{M}'\rangle\!\rangle : \mathbf{B}''$, which follows by the induction hypothesis.

Case: $\Gamma \vdash \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 : \mathbf{B}''[\mathbf{x} := \mathbf{V}]$ By the premise, we know:

(a) $\Gamma, \mathbf{V} = \mathbf{true} \vdash \mathbf{M}_1 : \mathbf{B}''[\mathbf{x} := \mathbf{true}]$ by inversion

(b) $\Gamma, \mathbf{V} = \mathbf{false} \vdash \mathbf{M}_2 : \mathbf{B}''[\mathbf{x} := \mathbf{false}]$ by inversion

(c) $\Gamma \vdash \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \texttt{hole}(\mathbf{M}_1)\ \mathbf{else}\ \texttt{hole}(\mathbf{M}_2) : \mathbf{B}[\mathbf{x} := \mathbf{V}]$ by definition

(d) $\Gamma \vdash \mathbf{K} : (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \texttt{hole}(\mathbf{M}_1)\ \mathbf{else}\ \texttt{hole}(\mathbf{M}_2) : \mathbf{B}[\mathbf{x} := \mathbf{V}]) \Rightarrow \mathbf{C}$ by definition

By [If], it suffices to show

(a) $\Gamma, \mathbf{V} = \mathbf{true} \vdash \mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{true}}\ \mathbf{x}][\mathbf{M}_1 // \mathbf{x}] : \mathbf{C}$

By definition, it suffices to show $\Gamma, \mathbf{V} = \mathbf{true} \vdash (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{true}}\ \mathbf{x}])\langle\!\langle\mathbf{M}_1\rangle\!\rangle : \mathbf{C}$.

By the induction hypothesis, it suffices to show that: $\Gamma, \mathbf{V} = \mathbf{true}, \texttt{defs}(\mathbf{M}_1) \vdash (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{true}}\ \mathbf{x}]) : (\texttt{hole}(\mathbf{M}_1) : \mathbf{B}''[\mathbf{x} := \mathbf{true}]) \Rightarrow \mathbf{C}$.

By continuation typing, it suffices to show

$\Gamma, \mathbf{V} = \mathbf{true}, \texttt{defs}(\mathbf{M}_1), \mathbf{x} = \texttt{hole}(\mathbf{M}_1) \vdash (\mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{true}}\ \mathbf{x}]) : \mathbf{C}$, which follows by Lemma 4.8 since

$\Gamma, \mathbf{V} = \mathbf{true}, \texttt{defs}(\mathbf{M}_1), \mathbf{x} = \texttt{hole}(\mathbf{M}_1) \vdash \mathbf{subst}_{\mathbf{V}=\mathbf{true}}\ \mathbf{x} \equiv \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \texttt{hole}(\mathbf{M}_1)\ \mathbf{else}\ \texttt{hole}(\mathbf{M}_2)$.

This final equivalence follows because $\mathbf{V}$ is either $\mathbf{true}$ (hence by reduction and [≡-Subst$_1$]) or $\mathbf{false}$ (hence by [≡-Absurd$_2$]) or a variable (hence by ζ reduction).

(b) $\Gamma, \mathbf{V} = \mathbf{false} \vdash \mathbf{K}[\mathbf{subst}_{\mathbf{V}=\mathbf{false}}\ \mathbf{x}][\mathbf{M}_2 // \mathbf{x}] : \mathbf{C}$, which follows symmetrically. □

5 ANF TRANSLATION

The ANF translation is presented in Figure 14. The naïve translation is defined inductively over syntax. The translation is indexed by a current continuation, which is used when translating a value and is is composed together "inside-out" the same continuation composition is defined in Section 4. The translation is essentially standard. When translating a value such as $\mathbf{x}$, $\lambda\,\mathbf{x} : \mathbf{A}.\,\mathbf{e}$, and $\mathbf{Type}_i$, we essentially plug the value into the current continuation, recursively translating the sub-expressions of the value if applicable. For non-values such as application, we make sequencing explicit by recursively translating each sub-expression with a continuation that binds the result which will perform the computation.

Note that if the translation must produce type annotations then defining the translation and typing preservation proof are somewhat more complicated. For instance, if we generate join points

$\boxed{[\![e]\!]\ K = M}$

$$
\begin{array}{rcl}
[\![e]\!] & \stackrel{\text{def}}{=} & [\![e]\!]\ [\cdot] \\
[\![x]\!]\ K & \stackrel{\text{def}}{=} & K[x] \\
[\![\mathsf{Prop}\ ]\!]\ K & \stackrel{\text{def}}{=} & K[\mathsf{Prop}\ ] \\
[\![\mathsf{Type}_i]\!]\ K & \stackrel{\text{def}}{=} & K[\mathsf{Type}_i] \\
[\![\Pi\, x : A.\, B]\!]\ K & \stackrel{\text{def}}{=} & K[\Pi\, x : [\![A]\!].\, [\![B]\!]] \\
[\![\lambda\, x : A.\, e]\!]\ K & \stackrel{\text{def}}{=} & K[\lambda\, x : [\![A]\!].\, [\![e]\!]] \\
[\![e_1\ e_2]\!]\ K & \stackrel{\text{def}}{=} & [\![e_1]\!]\ \mathsf{let}\ x_1 = [\cdot]\ \mathsf{in}\ [\![e_2]\!]\ \mathsf{let}\ x_2 = [\cdot]\ \mathsf{in}\ K[x_1\ x_2] \\
[\![\Sigma\, x : A.\, B]\!]\ K & \stackrel{\text{def}}{=} & K[\Sigma\, x : [\![A]\!].\, [\![B]\!]] \\
[\![\langle e_1, e_2\rangle\ \mathsf{as}\ A]\!]\ K & \stackrel{\text{def}}{=} & [\![e_1]\!]\ \mathsf{let}\ x_1 = [\cdot]\ \mathsf{in}\ [\![e_2]\!]\ (\mathsf{let}\ x_2 = [\cdot]\ \mathsf{in}\ K[(\langle x_1, x_2\rangle\ \mathsf{as}\ [\![A]\!])]) \\
[\![\mathsf{fst}\ e]\!]\ K & \stackrel{\text{def}}{=} & [\![e]\!]\ \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ K[\mathsf{fst}\ x] \\
[\![\mathsf{snd}\ e]\!]\ K & \stackrel{\text{def}}{=} & [\![e]\!]\ \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ K[\mathsf{snd}\ x] \\
[\![\mathsf{let}\ x = e\ \mathsf{in}\ e']\!]\ K & \stackrel{\text{def}}{=} & [\![e]\!]\ \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ [\![e']\!]\ K \\
[\![\mathsf{bool}]\!]\ K & \stackrel{\text{def}}{=} & K[\mathsf{bool}] \\
[\![\mathsf{true}]\!]\ K & \stackrel{\text{def}}{=} & K[\mathsf{true}] \\
[\![\mathsf{false}]\!]\ K & \stackrel{\text{def}}{=} & K[\mathsf{false}] \\
[\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\ K & \stackrel{\text{def}}{=} & [\![e]\!]\ \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ [\![e_1]\!]\ \mathsf{let}\ y = [\cdot]\ \mathsf{in}\ K[\mathsf{subst}_{x=\mathsf{true}}\ y] \\
 & & \quad \mathsf{else}\ [\![e_2]\!]\ \mathsf{let}\ y = [\cdot]\ \mathsf{in}\ K[\mathsf{subst}_{x=\mathsf{false}}\ y]
\end{array}
$$

Fig. 14. Naïve ANF Translation

directly, or require the **let**-bindings in the target language to have type annotations for bound expressions, then we would need to modify the translation to produce those annotations. This requires defining the translation over typing derivations, so the compiler has access to the type of the expression and not only its syntax.

Our goal is to prove type preservation: if e is well-typed in the source, then $[\![e]\!]$ is well-typed at a translated type in the target. But to prove type preservation, we must also preserve the rest of the judgmental and syntactic structure that dependent type systems rely on. To prove type-preservation, we follow a standard architecture for dependent type theory [Barthe et al. 1999; Barthe and Uustalu 2002; Boulier et al. 2017; Bowman and Ahmed 2018; Bowman et al. 2018]. Since type checking requires definitional equivalence, in the [Conv] rule, and substitution, in rules such as [App], we must preserve definitional equivalence and substitution. Since definitional equivalence is defined in terms of reduction, we must preserve reduction up to equivalence. Many of the proofs of lemmas are omitted for brevity and can be found in [12].

We stage the type-preservation proof as follows. First, we show *compositionality*, which states that the translation commutes with composition, *e.g.*, that substituting first and then translating is equivalent to translating first and then substituting. This proof is somewhat non-standard for ANF since the notion of composition in ANF is not the usual substitution. Next, we show that reduction and conversion are preserved up to equivalence. Note that for this theorem, we are interested in the conversion semantics used for definitional equivalence, not in the machine semantics used to evaluate ANF terms. Then, we show *equivalence preservation*: if two terms are definitionally equivalent in the source, then their translations are definitionally equivalent. Finally, we can show type preservation of the ANF translation, using continuation typing to express the inductive

[12] https://www.williamjbowman.com/downloads/anf-sigma-techrpt.pdf

invariant required for ANF. The continuation typing allows us to formally state type preservation in terms of the intuitive reason that type preservation should hold: because the definitions expressed by the continuation typing suffice to prove equivalence between a computation variable and the original depended-upon expression.

After proving type preservation, we prove correctness of separate compilation for the ANF machine semantics. This requires a notion of linking, which we define later in this section. This proof is straightforward from the meta-theory about the machine semantics proved in Section 4, and from equivalence preservation.

Recall from Section 4, we shift from the **target language font** to the source language font whenever we shift out of ANF, such as when we perform standard substitution or conversion. When the shift in font is not apparent, we use the language boundary term $\mathcal{ST}()$.

Before we proceed, we state a property about the syntactic form produced by the translation, in particular, that the ANF translation does produce syntax in ANF (Theorem 5.1). The proof is straightforward so we elide it.

Theorem 5.1 (ANF). *For all* $\mathtt{e}$ *and* $\mathtt{K}$, $[\![\mathtt{e}]\!]\,\mathtt{K} = \mathtt{M}$ *for some* $\mathtt{M}$.

As discussed in Section 4, composition in ANF is somewhat non-standard. Normally, we compose via substitution, so the compositionality property we want is $[\![\mathtt{e}[\mathtt{x} := \mathtt{e}']]\!] \equiv [\![\mathtt{e}]\!][\mathtt{x} := [\![\mathtt{e}']\!]]$, which says we can either compose then translate or translate then compose. However, most composition in ANF goes through continuations, not through substitution, since only values can be substituted in ANF. Our primary compositionality lemma (Lemma 5.2) tells us that we can either first translate a program $\mathtt{e}$ under continuation $\mathtt{K}$ and then compose it with a continuation $\mathtt{K}'$, or we can first compose the continuations $\mathtt{K}$ and $\mathtt{K}'$ and then translate $\mathtt{e}$ under the composed continuation. Note that this proof is entirely within ECC^A; there are no language boundaries.

Lemma 5.2 (Compositionality). $\mathtt{K}'\langle\!\langle\, [\![\mathtt{e}]\!]\,\mathtt{K}\rangle\!\rangle = [\![\mathtt{e}]\!]\,\mathtt{K}'\langle\!\langle\mathtt{K}\rangle\!\rangle$

Corollary 5.3. $\mathtt{K}\langle\!\langle\, [\![\mathtt{e}]\!]\,\rangle\!\rangle = [\![\mathtt{e}]\!]\,\mathtt{K}$

Next we show compositionality of the translation with respect to substitution (Lemma 5.4). While the proof relies on the previous lemma, this lemma is different in that substitution is the primary means of composition within the type system. We must essentially show that substitution is equivalent to composing via continuations. Since standard substitution does not preserve ANF, this lemma does not equate ECC^A terms, but ECC terms that have been transformed via ANF translation. We will again use language boundaries to indicate a shift from ANF to non-ANF terms. Note that this lemma relies on uniqueness of names.

Lemma 5.4 (Substitution). $[\![\mathtt{e}[\mathtt{x} := \mathtt{e}']]\!]\,\mathtt{K} \equiv \mathcal{ST}((([\![\mathtt{e}]\!]\,\mathtt{K})[\mathtt{x} := [\![\mathtt{e}']\!]])$

Next we show equivalence is preserved, in two parts. First we show that reduction is preserved up to equivalence, and then show conversion is preserved up to equivalence. The proofs are straightforward; intuitively, ANF is just adding a bunch of ζ-reductions.

Lemma 5.5. *If* $\Gamma \vdash \mathtt{e} \rhd \mathtt{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathtt{e}]\!] \equiv [\![\mathtt{e}']\!]$.

Next we show that conversion is preserved up to equivalence. Note that past work has a minor bug in the *proof* of the following lemma [Bowman and Ahmed 2018; Bowman et al. 2018], although it does not invalidate their *theorems*. The past proofs only account for transitivity of $\rhd^*$, but fail to account for the congruence rules. This is not a significant issue, since their translations are compositional and the congruence rules follow essentially from compositionality. We give the key cases of this proof to demonstrate the correct structure.

Lemma 5.6. *If* $\Gamma \vdash e \rhd^* e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$

Proof. By induction on the structure of $\Gamma \vdash e \rhd^* e'$.

Case: [Red-Refl], trivial.

Case: [Red-Trans], by Lemma 5.5 and the induction hypothesis.

Case: [Red-Cong-Let]

We have $\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e \rhd^* \mathsf{let}\ x = e_1\ \mathsf{in}\ e'$ and $\Gamma \vdash e \rhd^* e'$.

We must show that $[\![\Gamma]\!] \vdash [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e]\!] \equiv [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e']\!]$.

$$
\begin{aligned}
& [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e]\!] \\
= \ & [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ y[y := e]]\!] && (6)\\
\equiv \ & \mathcal{ST}([\![\mathsf{let}\ x = e_1\ \mathsf{in}\ y][y := [\![e]\!]]) && \text{by Lemma 5.4 (Substitution)} \quad (7)\\
\equiv \ & \mathcal{ST}([\![\mathsf{let}\ x = e_1\ \mathsf{in}\ y][y := [\![e']\!]]) && \text{by the induction hypothesis} \quad (8)\\
\equiv \ & [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ y[y := e']]\!] && \text{by Lemma 5.4} \quad (9)\\
= \ & [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e']\!] && (10)
\end{aligned}
$$

□

The previous two lemmas imply equivalence preservation. Including η-equivalence makes this non-trivial, but not hard.

Lemma 5.7. *If* $\Gamma \vdash e \equiv e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$

Since we implement cumulative universes through subtyping, we must also show subtyping is preserved (Lemma 5.8). The proof is completely uninteresting, except insofar as it is simple, while it seems to be impossible for CPS translation [Bowman et al. 2018]. We discuss this further in Section 7.

Lemma 5.8. *If* $\Gamma \vdash e \preceq e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \preceq [\![e']\!]$

We now prove type preservation, with a suitably strengthened induction hypothesis. We prove that, given a well-typed source term e of type A, and a continuation K that expects the definitions $\mathtt{defs}([\![e]\!])$, expects the term $\mathtt{hole}([\![e]\!])$, and has result type B, the translation $[\![e]\!]\,K$ is well typed.

The structure of the lemma and its proof are a little surprising. Intuitively, we would expect to show something like "if $e : A$ then $[\![e]\!] : [\![A]\!]$". We will ultimately prove this, Theorem 5.11 (Type Preservation), but we need a stronger lemma first (Lemma 5.10). Since the translation is pushing computation inside-out (since continuations compose inside-out), our type-preservation lemma and proof are essentially inside-out. Instead of the expected statement, we must show that if we have a continuation K that expects $[\![e]\!] : [\![A]\!]$, then we get a term $[\![e]\!]\,K$ of some arbitrary type B. In order to show that, we will have to show that $[\![e]\!] : [\![A]\!]$ and then appeal to Lemma 4.8 (Cut). Furthermore, each appeal to the inductive hypothesis will have to establish that we can in fact create well-typed continuations from the assumption that $[\![e]\!] : [\![A]\!]$.

Wielding our propositions-as-types hat, we can view this theorem as in accumulator-passing style, where the well-typed continuation is an accumulator expressing the inductive invariant for type preservation.

We begin with a minor technical lemma (Lemma 5.9) that will come in useful in the proof of type preservation. This lemma allows us to establish that a continuation is well typed when it expects an inductively smaller translated term in its hole. It also tells us, formally, that the inductive hypothesis implies the type preservation theorem we expect.

Lemma 5.9. *If for all* $\Gamma \vdash e : A$ *and* $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash K : (\mathtt{hole}([\![e]\!]) : [\![A]\!]) \Rightarrow B$ *we know that* $[\![\Gamma]\!] \vdash [\![e]\!]\,K : B$, *then* $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash \mathtt{hole}([\![e]\!]) : [\![A]\!]$ *(and, incidentally,* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$*)*

Lemma 5.10.

(1) If $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]$
(2) If $\Gamma \vdash e : A$, *and* $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash \mathbf{K} : (\mathtt{hole}([\![e]\!]) : [\![A]\!]) \Rightarrow \mathbf{B}$ *then* $[\![\Gamma]\!] \vdash [\![e]\!]\, \mathbf{K} : \mathbf{B}$

Proof. The proof is by induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash e : A$. The key cases are the typing rules that use dependency, that is, [Snd], [App], and [Let]. We give a couple representative cases; the other cases are essentially similar.

Case: [Ax-Prop]
We must show that $[\![\Gamma]\!] \vdash [\![\mathbf{Prop}\,]\!]\, \mathbf{K} : \mathbf{B}$.
By definition of the translation, it suffices to show that $[\![\Gamma]\!] \vdash \mathbf{K}[\mathbf{Prop}\,] : \mathbf{B}$.
Note that $\mathtt{defs}([\![\mathbf{Prop}\,]\!]) = \cdot$; this property holds for all values.
By Lemma 4.8 (Cut), it suffices to show that
(a) $\mathtt{hole}([\![\mathbf{Prop}\,]\!]) = \mathbf{Prop}$, which is true by definition of the translation, and
(b) $\mathbf{Prop} : [\![\mathbf{Type}_1]\!]$, which is true by [Ax-Prop], since $[\![\mathbf{Type}_1]\!] = \mathbf{Type}_1$.

Case: [If] This case is essentially similar to the **if** case for Lemma 4.12. Note that composing a continuation $\mathbf{K}$ with a configuration $\mathbf{M}$ has to perform the ANF translation for **if**.

Case: [Snd]
We must show $[\![\Gamma]\!] \vdash [\![\mathbf{snd}\, e]\!]\, \mathbf{K} : \mathbf{B}$, where $[\![\Gamma]\!], \mathtt{defs}([\![\mathbf{snd}\, e]\!]) \vdash \mathbf{K} : (\mathtt{hole}([\![\mathbf{snd}\, e]\!]) : [\![B'[x := \mathbf{fst}\, e]]\!]) \Rightarrow \mathbf{B}$ and $\Gamma \vdash e : \Sigma\, x : A'.\, B'$.
That is, by definition of the translation, we must show, $[\![\Gamma]\!] \vdash [\![e]\!]\, \mathbf{let}\, x' = [\cdot]\, \mathbf{in}\, \mathbf{K}[\mathbf{snd}\, x'] : \mathbf{B}$.
Let $\mathbf{K}' = \mathbf{let}\, x' = [\cdot]\, \mathbf{in}\, \mathbf{K}[\mathbf{snd}\, x']$.
Note that we know nothing further about the structure of the term we're trying to type check, $[\![e]\!]\, \mathbf{K}'$. Therefore, we cannot appeal to any typing rules directly. This happens because e is a computation, and the translation of computations composes continuations, which occurs "inside-out". Instead, our proof proceeds "inside-out": we build up typing invariants in a well-typed continuation $\mathbf{K}'$ (that is, we build up definitions in our accumulator) and then appeal to the induction hypothesis for e with $\mathbf{K}'$. Intuitively, some later case of the proof that knows more about the structure of e will be able to use this well-typed continuation to proceed.
So, by the induction hypothesis applied to $\Gamma \vdash e : \Sigma\, x : A'.\, B'$ with $\mathbf{K}'$, it suffices to show that:
$[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash \mathbf{let}\, x' = [\cdot]\, \mathbf{in}\, \mathbf{K}[\mathbf{snd}\, x'] : (\mathtt{hole}([\![e]\!]) : [\![\Sigma\, x : A'.\, B']\!]) \Rightarrow \mathbf{B}$
By [K-Bind], it suffices to show that
(a) $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash \mathtt{hole}([\![e]\!]) : [\![\Sigma\, x : A'.\, B']\!]$, which follows by Lemma 5.9 applied to the induction hypothesis for $\Gamma \vdash e : \Sigma\, x : A'.\, B'$
(b) $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), x' = \mathtt{hole}([\![e]\!]) \vdash \mathbf{K}[\mathbf{snd}\, x'] : \mathbf{B}$.
To complete this case of the proof, it suffices to show Item (b).
Note that $\mathtt{defs}([\![\mathbf{snd}\, e]\!]) = (\mathtt{defs}([\![e]\!]), x' = \mathtt{hole}([\![e]\!]))$ and $\mathtt{hole}([\![\mathbf{snd}\, e]\!]) = \mathbf{snd}\, x'$.
So, by Lemma 4.8 (Cut), given the type of $\mathbf{K}$, it suffices to show that
$[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), x' = \mathtt{hole}([\![e]\!]) \vdash \mathbf{snd}\, x' : [\![B'[x := \mathbf{fst}\, e]]\!]$.
By Lemma 5.4, $[\![B'[x := \mathbf{fst}\, e]]\!] \equiv [\![B']\!][x := [\![\mathbf{fst}\, e]]\!]$.
By [Conv], it suffices to show that $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), x' = \mathtt{hole}([\![e]\!]) \vdash \mathbf{snd}\, x' : [\![B']\!][x := [\![\mathbf{fst}\, e]]\!]$.
Note that we cannot show this by the typing rule [Snd], since the substitution $[\![B']\!][x := [\![\mathbf{fst}\, e]]\!]$ copies an apparently arbitrary expression $[\![\mathbf{fst}\, e]\!]$ into the type, instead of the expected subexpression $\mathbf{fst}\, x'$. That is, [Snd] tells us $\mathbf{snd}\, x' : [\![B']\!][x := \mathbf{fst}\, x']$ but we must show $\mathbf{snd}\, x' : [\![B']\!][x := [\![\mathbf{fst}\, e]]\!]$. The translation has disrupted the dependency on e, changing the type that depended on the specific value e into a type that depends on an apparently arbitrary value x'. This is the problem discussed in Section 2. It is also where the machine steps we have

$\boxed{v \approx V}$

$$\mathsf{true} \approx \mathbf{true} \qquad \mathsf{false} \approx \mathbf{false}$$

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

$$\frac{}{\cdot \vdash \emptyset} \qquad \frac{\cdot \vdash e : A \qquad \Gamma \vdash \gamma}{\Gamma, x = e \vdash \gamma[x \mapsto \gamma(e)]} \qquad \frac{\cdot \vdash e : A \qquad \Gamma \vdash \gamma}{\Gamma, x : A \vdash \gamma[x \mapsto e]}$$

Fig. 15. Separate Compilation Definitions

accumulated in our continuation typing save us. We can show that $[\![\mathsf{fst}\, e]\!] \equiv \mathbf{fst}\, x'$, under the definitions we have accumulated from continuation typing. This follows by Corollary 4.11. Therefore, by [Conv], it suffices to show that
$[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), x' = \mathtt{hole}([\![e]\!]) \vdash \mathbf{snd}\, x' : [\![B']\!][x := \mathbf{fst}\, x']$.
By [Snd], it suffices to show $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), x' = \mathtt{hole}([\![e]\!]) \vdash x' : \Sigma\, x : [\![A']\!].\, [\![B']\!]$, which follows since, as we showed in Item (a), $[\![e]\!] : \Sigma\, x : [\![A']\!].\, [\![B']\!]$. □

Theorem 5.11 (Type Preservation). *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$

We also prove correctness of separate compilation with respect to the ANF evaluation semantics. To do this we must define linking and define a specification, independent of the compiler, of when outputs are related across languages.

We first define observations, and when observations are related across languages. Without such a relation, the best we can prove is that the *translation* of the value v produced in the source is *definitionally equivalent* to the value we get by running the translated term, *i.e.*, we would get $[\![v]\!] \equiv \mathbf{eval}([\![e]\!])$. This fails to tell us how $[\![v]\!]$ is related to v, unless we inspect the compiler. Instead, we define an independent specification relating observation across languages, which allows us to understand the correctness theorem without reading the compiler. We define the relation $v \approx V$ to compare ground values in Figure 15.

We define linking as substitution with well-typed closed terms, and define a closing substitution γ with respect to the environment Γ (also in Figure 15). Linking is defined by closing a term e such that $\Gamma \vdash e : A$ with a substitution $\Gamma \vdash \gamma$, written $\gamma(e)$. Any γ is valid for Γ if it maps each $x : A \in \Gamma$ to a closed term e of type A. For definitions in Γ, we require that if $x = e \in \Gamma$, then $\gamma[x \mapsto \gamma(e)]$, that is, the substitution must map x to a closed version of its definition e. We lift the ANF translation to substitutions.

Correctness of separate compilation says that we can either link then run a program in the source language semantics, *i.e.*, using the conversion semantics, or separately compile the term and its closing substitution then run in the ANF evaluation semantics. Either way, we get equivalent terms.

Theorem 5.12 (Correctness of Separate Compilation). *If* $\Gamma \vdash e : A$, *(and* A *ground) and* $\Gamma \vdash \gamma$ *then* $\mathbf{eval}([\![\gamma]\!]\,([\![e]\!])) \approx \mathrm{eval}(\gamma(e))$.

Proof. The following diagram commutes, because $\equiv$ corresponds to $\approx$ on ground types, the translation commutes with substitution, preserves equivalence, reduction implies equivalence, and equivalence is transitive.

$$\begin{array}{ccc} \mathrm{eval}(\gamma(e)) & \xrightarrow{\equiv} & [\![\gamma(e)]\!] \\ \downarrow{\scriptstyle \equiv} & & \downarrow{\scriptstyle \equiv} \\ \mathbf{eval}([\![\gamma([\![e]\!])]\!]) & \xrightarrow{\equiv} & [\![\gamma([\![e]\!])]\!] \end{array}$$

□

6 JOIN-POINT OPTIMIZATION

Recall from Figure 9 that the composition of a continuation $\mathbf{K}$ with a configuration $\mathbf{if\ V\ then\ M_1\ else\ M_2}$, $\mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle$, duplicates $\mathbf{K}$ in the branches: intuitively, $\mathbf{if\ V\ then\ K}\langle\!\langle \mathbf{M_1} \rangle\!\rangle\ \mathbf{else\ K}\langle\!\langle \mathbf{M_2} \rangle\!\rangle$, although for type checking we need the ANF version of $\mathbf{K}\langle\!\langle \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{M_1} \rangle\!\rangle$ for the branches. Similarly, the ANF translation in Figure 14 performs the same duplication when translating **if** expressions. This can cause exponential code duplication, which is no problem in theory but is a problem in practice.

ECC^A supports implementing the join-point optimization, which avoids this code duplication. We can see the optimization as either a modification to the ANF translation itself, or as an intra-language optimization over ECC^A programs. We present join-point optimization as the latter, as it is strictly more general (applies to all ECC^A terms, instead of only terms generated by one compiler) and easier to prove correct.

Formally, in ECC^A, the join-point optimization requires the following equivalence.

$$\begin{aligned}\mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle \equiv\ & \mathbf{let\ f} = (\lambda\, \mathbf{x} : \mathbf{B}[\mathbf{x} := \mathbf{V}].\ \mathbf{K}[\mathbf{x}])\ \mathbf{in} \\ & \mathbf{if\ V\ then}\ (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x\ in\ f\ z})\langle\!\langle \mathbf{M_1} \rangle\!\rangle \\ & \mathbf{else}\ (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=false}}\ \mathbf{x\ in\ f\ z})\langle\!\langle \mathbf{M_2} \rangle\!\rangle\end{aligned}$$

Note that instead of duplicating $\mathbf{K}$ in the branches as before, we introduce a function $\mathbf{f}$ that abstracts over $\mathbf{K}$ and call it in the branches. In essence, this introduces a local first-class continuation $\mathbf{f}$ that abstracts the non-first-class continuation $\mathbf{K}$. We also prove a more general statement of this equivalence in our Coq model in [13].

From β and ζ reduction, it is simple to conclude the desired equivalence, formally stated below as Lemma 6.1.

Lemma 6.1 (Join Point Equivalence).
$$\begin{aligned}\mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle \equiv\ & \mathbf{let\ f} = (\lambda\, \mathbf{x} : \mathbf{B}[\mathbf{x} := \mathbf{V}].\ \mathbf{K}[\mathbf{x}])\ \mathbf{in} \\ & \mathbf{if\ V\ then}\ (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x\ in\ f\ z})\langle\!\langle \mathbf{M_1} \rangle\!\rangle \\ & \mathbf{else}\ (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=false}}\ \mathbf{x\ in\ f\ z})\langle\!\langle \mathbf{M_2} \rangle\!\rangle\end{aligned}$$

The more difficult part is showing that the join point optimization is well typed. To do this, we need continuation typing. Assuming that the unoptimized term $\mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle$ is well typed and $\mathbf{K}$ itself has the expected type, then the join-point optimization is well typed.

Lemma 6.2 (Join Point Well Typed). *If* $\Gamma \vdash \mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle : \mathbf{C}$, $\Gamma \vdash \mathbf{if\ V\ then\ M_1\ else\ M_2} : \mathbf{B}[\mathbf{x} := \mathbf{V}]$, *and* $\Gamma \vdash \mathbf{K} : \mathtt{hole}(\mathbf{if\ V\ then\ M_1\ else\ M_2}) : \mathbf{B'} \Rightarrow \mathbf{C}$ *then*
$$\begin{aligned}&\Gamma \vdash \mathbf{let\ f} = (\lambda\, \mathbf{x} : \mathbf{B}[\mathbf{x} := \mathbf{V}].\ \mathbf{K}[\mathbf{x}])\ \mathbf{in} \\ &\quad \mathbf{if\ V\ then}\ (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x\ in\ f\ z})\langle\!\langle \mathbf{M_1} \rangle\!\rangle \\ &\quad\quad \mathbf{else}\ (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=false}}\ \mathbf{x\ in\ f\ z})\langle\!\langle \mathbf{M_2} \rangle\!\rangle : \mathbf{C}\end{aligned}$$

Proof. By definition, $\Gamma \vdash \mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle : \mathbf{C}$ implies
$\Gamma \vdash \mathbf{if\ V\ then\ K}[\mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x}][\mathbf{M_1}/\!/\mathbf{x}]\ \mathbf{else\ K}[\mathbf{subst}_{\mathbf{V=false}}\ \mathbf{x}][\mathbf{M_2}/\!/\mathbf{x}] : \mathbf{C}$
Let $\Gamma' = (\Gamma, \mathbf{f} = \lambda\, \mathbf{x} : (\mathbf{B}[\mathbf{x} := \mathbf{V}]).\ \mathbf{K}[\mathbf{x}])$, and note that $\mathbf{f} : (\mathbf{B}[\mathbf{x} := \mathbf{V}]) \to \mathbf{C}$.
By [If] and Lemma 4.12 (Heterogeneous Cut), it suffices to show that

(1) $\Gamma', \mathbf{V} = \mathbf{true}, \mathtt{defs}(\mathbf{M_1}) \vdash (\mathbf{let\ x} = [\cdot], \mathbf{z} = \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x\ in\ f\ z}) : (\mathtt{hole}(\mathbf{M_1}) : \mathbf{B}[\mathbf{true} := \mathbf{x}]) \Rightarrow \mathbf{C}$
By [K-Bind] and [Let], it suffices to show
$\Gamma', \mathbf{V} = \mathbf{true}, \mathtt{defs}(\mathbf{M_1}), \mathbf{x} = \mathtt{hole}(\mathbf{M_1}), \mathbf{z} = \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x} \vdash \mathbf{f\ z} : \mathbf{C}$
By [App], it suffices to show that
$\Gamma', \mathbf{V} = \mathbf{true}, \mathtt{defs}(\mathbf{M_1}), \mathbf{x} = \mathtt{hole}(\mathbf{M_1}) \vdash \mathbf{subst}_{\mathbf{V=true}}\ \mathbf{x} : (\mathbf{B}[\mathbf{x} := \mathbf{V}])$,
which follows by [Subst] since $(\mathtt{hole}(\mathbf{M_1}) : \mathbf{B'}[\mathbf{x} := \mathbf{true}])$ and we have $\mathbf{V} = \mathbf{true}$.

[13] https://www.williamjbowman.com/downloads/anf-sigma-techrpt.pdf

$\boxed{[\![e]\!]\ K = M}$

$$
\begin{array}{rcl}
[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!]\ K & \stackrel{\text{def}}{=} & [\![e]\!]\ \text{let } x' = [\cdot] \text{ in let } f = (\lambda\, x : [\![B]\!][x := [\![e]\!]].\ K[x]) \text{ in} \\
& & \quad \text{if } x' \text{ then } [\![e_1]\!]\ (\text{let } x = [\cdot], z = \text{subst}_{x'=\text{true}}\ x \text{ in } f\ z) \\
& & \quad \text{else } [\![e_2]\!]\ (\text{let } x = [\cdot], z = \text{subst}_{x'=\text{false}}\ x \text{ in } f\ z) \\
& & \quad \text{where } \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : B[x := e]
\end{array}
$$

Fig. 16. Join-Point Optimized ANF Translation

(2) $\Gamma', V = \text{false}, \texttt{defs}(M_2) \vdash (\text{let } x = [\cdot], z = \text{subst}_{V=\text{false}}\ x \text{ in } f\ z) : (\texttt{hole}(M_2) : B[\text{false} := x]) \Rightarrow C$, which follows symmetrically. □

It's a simple corollary to show we can now modify the ANF translation from Section 5 to use the definition in Figure 16, and it is still correct and type preserving. However, note that the new translation requires access to the type B from the derivation. We can do this either by defining the translation by induction over typing derivations, or (preferably) modifying the syntax of if to include B as a type annotation, similar to dependent pairs.

7 RELATED WORK

7.1 CPS Translation

ANF is favored as a compiler intermediate representation, although not universally. Maurer et al. [2017] argue for ANF, over alternatives such as CPS, because ANF makes control flow explicit but keeps evaluation order implicit, automatically avoids administrative redexes, simplifies many optimizations, and keeps code in direct style. Kennedy [2007] argues the opposite—that CPS is preferred to ANF—primarily due to the complexity of the composition operations and join points, and summarizes the arguments for and against.

Most recent work on CPS translation of dependently typed languages is focused on expressing control effects. Pédrot [2017] uses a non-standard CPS translation to internalize classical reasoning in the Calculus of Inductive Constructions (CIC). Cong and Asai [2018a,b] develop CPS translations to express delimited control effects, via `shift` and `reset`, in a dependently typed language. Miquey [2017] uses a CPS translation to model a dependent sequent calculus. When expressing control effects in dependently typed languages, it is *necessary* to prevent certain dependencies from being expressed to maintain consistency [Barthe and Uustalu 2002; Herbelin 2005], therefore these translations do not try to recover dependencies in the way we discuss in Section 2.

The CPS translation of Bowman et al. [2018] is a type-preserving translation for the Calculus of Constructions. They add a special form and typing rule for the application of a computation to a continuation which essentially records a machine step. Unfortunately, this rule relies on interpreting all functions as parametric, and their type translation does not scale to higher universes. Formally, preservation of subtyping, Lemma 5.8 does not hold when extending their CPS translation to a language with higher universes. By contrast, our ANF translation works with higher universes and, our model of ECC^A is orthogonal to parametricity.

Cong and Asai [2018a] extend the translation of Bowman et al. [2018] to dependent pattern matching using essentially the same typing rule for **if** as we do in ECC^A. Their translation is also unable to scale to higher universes.

7.2 Call-By-Push-Value

Attempts to add dependent types to call-by-push-value (CBPV) face a similar challenge to the one we discuss in Section 2, because CBPV linearize some expressions to make sequences of effects explicit [Ahman 2017; Vákár 2017]. However, CBPV is in monadic form, which allows CBPV to

ignore the central problem in the ANF transformation. Nested **let**s are permitted, so the dependent **let** without definitions suffices [Vákár 2017]. On the other hand, CBPV is also meant to support effects, which add a new challenge, since a dependent elimination could copy an effectful subexpression in to a type, thus duplicating the effect. Vákár [2017] notes that adding dependent let (called in that work the dependent Klesli extension) makes it impossible to model many effects.

We conjecture dependent CBPV could be modified to support dependent let, ANF, and effects. This would require adding the machine steps from the typing rules presented in Section 2 to CBPV, but also adding support for interpreting machine steps as values. In our work, we rely on the fact that every computation is pure, so that encoding of machine steps like $\mathbf{x} = \mathbf{M}$ and $\mathbf{M} = \mathbf{true}$ can be easily interpreted in the type system. In CBPV, machine steps can be effectful, so we would also need to internalize the fact that a computation $\mathbf{M}$ actually behaves purely and so can be interpreted as a value. So far, this has not been done in dependent CBPV, although a recent draft by Pédrot and Tabareau [2017] is quite close to a solution. They provide types for "thunkable" computations, which are effectful computations whose effects are bracketed allowing the computation to be suspended and copied.

7.3 Commutative Cuts

As mentioned in Section 2, the general problem that certain transformations do not preserve dependent typing is the problem of commutative cuts [Boutillier 2012; Herbelin 2009]. Recall from Section 2 that we wanted the following terms in ECC to be equivalent and have the same type.

$$\mathsf{f}\ (\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2) \equiv \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{f}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{f}\ \mathsf{e}_2$$

This does not hold, but in ECC^A the following essentially similar equivalence holds.

$$\mathbf{f}\ (\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2) \equiv \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{f}\ (\mathbf{subst}_{\mathbf{e}=\mathbf{true}}\ \mathbf{e}_1)\ \mathbf{else}\ \mathbf{f}\ (\mathbf{subst}_{\mathbf{e}=\mathbf{false}}\ \mathbf{e}_2)$$

Our solution in ECC^A works for ANF, but a different solution is required for commutative cuts in general. Our proofs in ANF relied on a key restriction: that all our proofs use continuation typing, with continuations of types $\mathbf{K} : (\mathbf{M} : \mathbf{B}) \Rightarrow \mathbf{C}$, and, critically, that $\mathbf{C}$ *cannot depend on* $\mathbf{M}$. We relied on this property even in Section 2, giving f a *non-dependent* function type $\mathsf{B}[\mathsf{x} := \mathsf{e}] \to \mathsf{C}$. If f instead has a dependent type $\Pi\, \mathsf{x}' : (\mathsf{B}[\mathsf{x} := \mathsf{e}]).\ \mathsf{C}$, then we need to also establish two additional type equivalences:

$$\mathsf{e} = \mathsf{true} \vdash \mathsf{C}[\mathsf{x}' := (\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)] \equiv \mathsf{C}[\mathsf{x}' := \mathsf{e}_1]$$

$$\mathsf{e} = \mathsf{false} \vdash \mathsf{C}[\mathsf{x}' := (\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)] \equiv \mathsf{C}[\mathsf{x}' := \mathsf{e}_2].$$

These require additional coercions for the applications $\mathsf{f}\ \mathsf{e}_1$ and $\mathsf{f}\ \mathsf{e}_2$ in the branches. For commutative cuts in general, we require the following equivalence and both sides to have the same type. This is true in ECC^A.

$$\begin{aligned}\mathbf{f}\ (\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2) \equiv\ & \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{subst}_{\mathbf{e}=\mathbf{true}}\ (\mathbf{f}\ (\mathbf{subst}_{\mathbf{e}=\mathbf{true}}\ \mathbf{e}_1)) \\ & \mathbf{else}\ \mathbf{subst}_{\mathbf{e}=\mathbf{false}}\ (\mathbf{f}\ (\mathbf{subst}_{\mathbf{e}=\mathbf{false}}\ \mathbf{e}_2))\end{aligned}$$

$$\text{where } \mathbf{f} : \mathbf{\Pi}\, \mathbf{x}' : \mathbf{B}[\mathbf{x} := \mathbf{e}].\ \mathbf{C} \text{ and } (\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2) : \mathbf{B}[\mathbf{x} := \mathbf{e}]$$

Note that the type of $\mathbf{C}$ cannot be dependent, so no further equivalence can be required.

At least one additional problem remains with commutative cuts, namely how they interact with the syntactic guard condition. Boutillier [2012] gives a relaxation of Coq's guard condition that admits some commutative cuts, but does not prove normalization. We have not yet considered how the guard condition interacts with ECC^A.

8 CONCLUSION

We develop a type-preserving ANF translation for ECC—a significant subset of Coq including dependent functions, dependent pairs, dependent elimination of booleans, and the infinite hierarchy

of universes—and prove correctness of separate compilation with respect to a machine semantics for the target language. This translation provides strong evidence that type-preservation compilation can support all of dependent type theory, gives insights into type preservation for dependent in general, into past work on type-preserving control flow transformations, and into combining effects and dependently typed languages like in dCBPV. The translation should scale to languages such as Coq, and be useful as the first pass of a type-preserving compiler for Coq.

ACKNOWLEDGMENTS

We gratefully acknowledge Youyou Cong, Max S. New, Hugo Herbelin, Matthias Felleisen, Greg Morrisett, Simon Peyton Jones, Paul Downen, Andrew Kennedy, Brian LaChance, and Danel Ahman for their time in discussing ANF, CPS and related problems during the course of this work. For real, thank you all.

REFERENCES

Danel Ahman. 2017. *Fibred Computational Effects.* Ph.D. Dissertation. University of Edinburgh. http://arxiv.org/abs/1710.02594

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

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

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

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 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

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

Pierre Boutillier. 2012. A relaxation of Coq's guard condition. In *Journées Francophones des langages applicatifs (JFLA).* https://hal.archives-ouvertes.fr/hal-00651780

William J. Bowman and Amal Ahmed. 2018. Typed Closure Conversion for the Calculus of Constructions. In *International Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3192366.3192372

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

Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. http://adam.chlipala.net/cpdt/

Youyou Cong and Kenichi Asai. 2018a. Handling Delimited Continuations with Dependent Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, ICFP (Sept. 2018). https://doi.org/10.1145/3236764

Youyou Cong and Kenichi Asai. 2018b. Shifting and Resetting in the Calculus of Constructions. In *International Symposium on Trends in Functional Programming (TFP).* https://sites.google.com/site/youyoucong212/tfp2018

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2-3 (Feb. 1988). https://doi.org/10.1016/0890-5401(88)90005-3

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

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

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Symposium on Operating Systems*

Design and Implementation (OSDI). https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

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. 2009. On a few open problems of the Calculus of Inductive Constructions and on their practical consequences. pauillac.inria.fr/~herbelin/talks/cic.pdf Updated 2010.

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

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

Zhaohui Luo. 1990. *An Extended Calculus of Constructions*. Ph.D. Dissertation. University of Edinburgh. http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-118/

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon 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

George C. Necula. 1997. Proof-Carrying Code. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/263699.263712

Pierre-Marie Pédrot. 2017. A Parametric CPS to Sprinkle CIC with Classical Reasoning. In *Workshop on Syntax and Semantics of Low-Level Languages*. http://www.cs.bham.ac.uk/~zeilbern/lola2017/abstracts/LOLA_2017_paper_5.pdf

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. The Fire Triangle: How To Mix Substitution, Dependent Elimination and Effects. (2017). https://www.pe%CC%81drot.fr/articles/dcbpv.pdf

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. In *LISP and Functional Programming (LFP)*. https://doi.org/10.1145/141478.141563

Amr Sabry and Philip Wadler. 1997. A Reflection on Call-by-Value. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (Nov. 1997). https://doi.org/10.1145/267959.269968

Paula Severi and Erik Poll. 1994. Pure Type Systems with Definitions. In *International Symposium Logical Foundations of Computer Science (LFCS)*. https://doi.org/10.1007/3-540-58140-5_30

Hayo Thielecke. 2003. From Control Effects to Typed Continuation Passing. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/640128.604144

Matthijs Vákár. 2017. *In Search of Effectful Dependent Types*. Ph.D. Dissertation. Oxford University. http://arxiv.org/abs/1706.07997

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2003. A Concurrent Logical Framework: The Propositional Fragment. In *International Workshop on Types for Proofs and Programs (TYPES)*. https://doi.org/10.1007/978-3-540-24849-1_23