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 get no guarantees after compiling or 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 unverified 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 checking dependent types relies strongly on the syntax of programs, which compilers change.

We extend the A-normal form (ANF) translation—a standard translation for making control flow explicit—to the Extended Calculus of Constructions (ECC), a representative subset of Coq. We prove type preservation, and prove correctness of separate compilation from ECC to an ANF-restricted 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), develop the program, and prove that the program meets its specification. These languages, and Coq in particular, have been widely used to develop formally verified high-assurance software including the CompCert C compiler [25], the CertiKOS operating system kernel [19, 20], and cryptographic primitives [4] and protocols [6].

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 unknown code C to form the whole program P. If C violates some specification of S, then we say a linking error occurs and then P contains safety, security, or correctness errors, even if S was compiled with a verified compiler. We can develop simple examples in Coq that, once compiled to OCaml and linked with an unverified OCaml component, jump to an arbitrary location in memory—despite the Coq program being proven memory safe with a machine-checked proof.

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. Ongoing work on CertiCoq [3] seeks to develop a verified compiler for Coq, but it cannot rule out linking with unsafe target code.

We could rule out linking errors with type-preserving compilation. A type-preserving compiler for Coq would preserve dependent types, which represent specifications, from the source language

*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.
down to a dependently typed intermediate language (IL). We would then use type checking at link time to enforce specifications when linking with unknown code, essentially implementing proof-carrying code [30]. In addition to ruling out linking errors, preserving specifications encoded in dependent types would let us implement certifying compilation [31] by type checking. To implement safe linking between the typed IL and standard (untyped) assembly, we could use gradual typing to enforce safety at the boundary between the typed IL and untyped components [2]. After linking in the IL, we would have a whole program, so it would be safe to erase types and used verified compilation to machine code. 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. For example, the standard type-preserving continuation-passing style (CPS) translation can result in undecidable type checking [7], and was shown impossible, in general, while maintaining consistency [8]. All recent work studying control and dependent types requires non-standard translations or type theories, or fails to scale to all of the core features of languages such as Coq [29, 32, 13, 15, 16]. Many of these translations model control effects, which is orthogonal to type preservation, so non-standard translations or theories may be acceptable in that setting; we discuss this further in Section 7. However, these are undesirable for a type-preserving compiler whose goal is to translate Coq.

The prevalence of results with undesirable properties—undecidability, inconsistency, non-standardness, reliance on strong axioms—leaves us with a pretty fundamental question: can we even do type-preserving compilation for dependent types, in theory? So far, the answer has been “yes, with great difficulty and by giving up on anything practical”. Supporting each new feature of dependency—the core theoretical features of dependent types used in practice—has introduced some undesirable property. Barthe et al. [7] were able to extend the standard type-preserving CPS translation to support $\Pi$ types, the most basic dependent type, but only at the expense of decidable type checking. Barthe and Uustalu [8] extend this further, to support $\Sigma$ types (another basic and necessary feature), but only by giving up consistency. Bowman et al. [13] manage to support both $\Pi$ and $\Sigma$ 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).

Before we can hope to build a verified, type-preserving compiler, capable of ruling out miscompilation and linking errors, and preserving our full functional verification to assembly code, we need a theory for dependent-type preserving translation. To have a hope of being practical, i.e., of scaling to a language such as Coq, this theory must support $\Pi$, $\Sigma$, higher universes, dependent case analysis, decidable type checking, consistency, and all the axioms that Coq users might rely on in their components. (Inductive types can be encoded using standard 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 of the above features?

In this paper, we develop a type-preserving translation for a significant subset of Coq, specifically the Extended Calculus of Constructions (ECC), to A-normal form (ANF). The focus of this work is supporting higher universes and avoiding parametricity and impredicativity, which are primary limitations in related work [13]. ECC includes all but one (dependent case analysis) of the above desiderata; in Section 7, we show how to apply past work to support dependent case analysis. This provides substantial evidence that dependent-type preservation can, in theory, scale to practice.

---

1In this paper, we include the key definitions and proof cases. Extended figures and proofs are available in the anonymous supplementary technical appendix.
The ANF translation is used to make control flow explicit in program syntax [18], particularly for lazy languages [28], and ANF is valued as an intermediate form for compiler optimizations [28, 24]. Our translation targets an ANF-restricted subset of ECC, which is consistent, has decidable type checking, and has a separate machine evaluation semantics. We also prove correctness of separate compilation with respect to the machine semantics. The translation and type theory of both the source and target are standard, unlike prior type-preserving translations, ensuring the translation works for existing dependently typed languages, and that we can reuse existing work on ANF translation.

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 6.

2 MAIN IDEAS
A-normal form (ANF) is a syntactic form that makes control flow explicit in the syntax of a program [33, 18]. ANF encodes computation (e.g., reducing an expression to a value) as sequencing 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 e1 e2 and we want a call-by-value semantics, we say the language first evaluates e1 to a value, then evaluates e2 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, as in let x0 = N0, ..., xn = Nn in N, where each xi = Ni is a primitive machine step: either move a value into xi, or perform a primitive computation (whose operands are values) and bind the result in xi. The ANF translation of e1 e2 is the following.

\[
\begin{align*}
\text{let } & x_0 = N_{1_0}, \ldots, x_n = N_{1_n}, x_1 = N_1 \\
& x_2 = N_{2_0}, \ldots, x_2_n = N_{2_n}, x_2 = N_2 \\
& \text{in } (x_1 \ x_2)
\end{align*}
\]

where \[e_1\] = (let x0 = N10, ..., x1n = N1n in N1) and \[e_2\] = (let x0 = N20, ..., x2n = N2n in N2).

Roughly, we can think of this as let x1 = \[e_1\], x2 = \[e_2\] in x1 x2. This rough translation is only ANF if e1 and e2 are values or primitive computations. In general, the ANF translation reassociates all the intermediate computations from \[e_1\] and \[e_2\] so there are no nested let expressions. Once in ANF, it is simple to formalize a machine semantics to implement evaluation by always reducing the left-most let-binding, which will be a primitive operation with values for operands. For a lazy semantics, we can instead delay each of the let-bindings and begin forcing the inner-most body (right-most expression).

The problem in developing a type-preserving translation for a dependently typed language is that changing the structure of a program 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. This pattern happens in dependent elimination forms, such as application and projection. For example, for a dependent function e1 : Πx : A. B, the application is typed as e1 e2 : B[e2/x]. Notice that the depended upon sub-expression e2 is copied into the type. If we transform the expression e1 e2, we can easily change the type. For example, recall the ANF translation for this expression given earlier.
The overall type of this expression above is roughly $[[B]][x_2/x] [[N_{2_0}/x_2]][...][N_{2_n}/x_2] [[N_{2}/x_2]]$. That is, the type $[[B]][x_2/x]$ of the body $x_1 x_2$ with all $\text{let}$-bindings substituted in. The type is due to the typing rule for dependent $\text{let}$ which substitutes $\text{let}$-expressions into the type of the body of the $\text{let}$, i.e., because $\text{let} z = e \text{ in } e' : B'[e/z]$ when $e : A'$ and $e' : B'$. To show type preservation, we must show that in the target type system, the above type of the ANF translated expression is equivalent to the translation of the original type $[[B]][e_2/x]]$. For decidable type checking, we must ensure this equivalence is decidable. Intuitively, it ought to be true that the intermediate computations $(x_{2_0} = N_{2_0}, ... , x_{2_n} = N_{2_n}, x_2 = N_2)$ are equivalent to $[[e_2]],$ since those are the steps that the machine must take to compute the value of $e_2$. Therefore, the two types ought to be equivalent.

But this intuitive argument that is wrong. Consider the expression below in which $(x_1 x_2)$ is $\text{let}$-bound and then used, instead appearing in the body directly as above.

$$\text{let} \ldots \text{in} y = (x_1 x_2) \text{ in } f \ y \quad \text{where } f : ([[B]][[[e_2]/x]]) \rightarrow C$$

To show type preservation, we now need to re-establish a dependent type that is $\text{let}$-bound, instead of in the body of a $\text{let}$. The type derivation fails, as follows.

$$\begin{align*}
\Gamma & \vdash \text{let } x = (x_1 x_2) \text{ in } y : C \\
\Gamma & \vdash f \ y : C
\end{align*}$$

This fails since $f$ expects $y : ([[B]][[[e_2]/x]])$, but is applied to $y : [[B]][x_2/x]$. We cannot show the two types are equal without substituting all the (elided) bindings as described earlier, but that substitution happens after it is needed, further down in the derivation tree. The problem is that the dependent typing rule for $\text{let}$ only binds a depended-upon expression in the type of the body after type checking the body, not while type checking.

The typing rule for $\text{let}$ essentially forgets that, by the time $f \ y$ happens, the machine will have performed the steps of computation ... $x_2 = N_2$. If we could record those machine steps in the derivation while type checking, instead of in the type after type checking, we can re-establish the dependency. For example, if we could express something like ...$x_2 = [[e_2]] \vdash (x_1 x_2) : [[B]][[[e_2]/x]],$ then we could complete the derivation and prove type preservation. And this is exactly the intuition we formalize.

We formalize this intuition using definitions [35], a standard extension to type theory that changes the typing rule for $\text{let}$ to thread equalities into sub-derivations and resolve dependencies. The typing rule for $\text{let}$ with definitions is:

$$\Gamma \vdash e : A \quad \Gamma, x = e \vdash e' : A' \quad \Gamma \vdash \text{let } x = e \text{ in } e' : A'[e/x]$$

The definition $x = e$ is introduced when type checking the body of the $\text{let}$, and can be used to solve type equivalence in sub-derivations, instead of only in the substitution $A'[e/x]$ 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) [35], and is a feature already found in dependently typed languages such as Coq. The admissibility means it’s uninteresting from a theoretical perspective, but it’s important to a compiler IL where syntactic form is meaningful.

Using definitions, we can prove that, under the definitions $x_{2_0} = N_{2_0}, ... , x_{2_n} = N_{2_n}, x_2 = N_2$ produced from the ANF translation $[e_2]$ (written $\text{defs}([e_2])$), the desired equivalence $x_2 \equiv [e_2]$ holds.
We use \( \text{hole}[e_2] \) to refer to the inner-most computation produced by the ANF translation, in this case \( \text{hole}[e_2] = N_2 \). Then we can prove \( \text{defs}[e_2] \vdash \text{hole}[e_2] = [e_2] \) (formally captured by Lemma 4.5). Since we also have the definition, \( x_2 = N_2 \), we conclude that \( \text{defs}[e_2] \vdash x_2 = [e_2] \).

In ANF, definitions are sufficient to encode machine steps in typing derivations, and the representation of machine steps as definitions makes deciding type equivalence easy. 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, and the general lesson is that future type preserving translations should carefully consider how to encode and record machine steps to ensure the translation will scale. In Section 6, we explain related work in terms of machine steps and compare to the encoding and typing rule above.

To formalize our type preservation argument, we need to step back and define the ANF translation precisely. In the source, looking at an expression such as \( e_1 e_2 \), we do not know whether the expression is embedded in a larger context. This matters in ANF, since we can no longer compose expressions by nesting, but instead must compose expressions with \( \text{let} \). This is why the above example translation disassembled the translation of \( e_1 \) and \( e_2 \) into a new, larger, \( \text{let} \) expression. To formalize the ANF translation, it helps to have a more compositional syntax for translating an expression and composing it with an unknown context.

To make it compositional, we index the ANF translation by a target language (non-first-class) continuation \( K \) representing the rest of the computation in which a translated expression will be used.\(^2\) A continuation \( K \) is a program with a hole (single linear variable) \([\cdot]\), and can be composed with a computation, written \( K[N] \), to form a program \( M \). In ANF, there are only two continuations: either \([\cdot]\) or \( \text{let} x = [\cdot] \text{ in } M \). Using continuations, we define ANF translation for functions and application as follows.

\[
\begin{align*}
[\lambda x : A. e] K &= K[\lambda x : ([A] \ [\cdot]). ([e] \ [\cdot])] \\
[e_1 e_2] K &= [e_1] \text{ let } x_1 = [\cdot] \text{ in } [e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2]
\end{align*}
\]

This allows us to focus on composing the primitive operations instead of reassociating \( \text{let} \) bindings.

We give typing rules to continuations to simplify the type preservation proof. The key typing rule is the following.

\[
\Gamma \vdash N : A \quad \Gamma, y = N \vdash M : B \\
\Gamma \vdash \text{let } y = [\cdot] \text{ in } M : (N : A) \Rightarrow B \quad \text{[K-Bind]}
\]

The type \( (N : A) \Rightarrow B \) of continuations describes that the continuation must be composed with the term \( N \) of type \( A \), and the result will be of type \( B \). This expresses syntactically the intuition that continuations must be used linearly to avoid control effects, which are known to cause inconsistency with dependent types \( [8, 21] \). Note that this type allows us to introduce the definition \( y = N \) via the type, before we know how that the continuation is used.\(^3\) We discuss this rule further in Section 4. The Lemma 4.4 (Cut) expresses that continuation typing is not an extension to the target type theory, which is important to ensure ANF can be applied in practice.

The key lemma to prove type preservation is the following.

**Lemma 2.1.** If \( \Gamma \vdash e : A \) and \( \text{defs}[e] \vdash K : (\text{hole}[e] : [A]) \Rightarrow B \), then \( \Gamma \vdash [e] K : B \).

This lemma captures the fact that each time we build a new \( K \) in the ANF translation, we must show it is well-typed, and that is where we apply the reasoning about definitions. Proving that \( K \) has the above type requires proving \( \Gamma, \text{defs}[e] \vdash \text{hole}[e] : [A] \). For our running example, this means proving

\(^2\) This how ANF translation is implemented in Scheme by Flanagan et al. \([18]\), although their mathematical presentation is as a reduction system.

\(^3\) 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.
The document discusses the syntax and semantics of the Extended Calculus of Constructions (ECC), a type system that includes dependent types and universes. The syntax includes terms, types, and environments, and the semantics involve rules for conversion and evaluation. The paper notes that the meta-variables are used to classify terms, and it mentions the decidability of type checking in ECC, which simplifies the ANF (Abstraction Normal Form) translation. The syntax and semantics are formalized with lambda calculus, dependent types, and universes, allowing for complex term evocation and type checking.

3 SOURCE: ECC WITH DEFINITIONS

Our source language, ECC, is Luo’s Extended Calculus of Constructions (ECC) [27] extended with definitions [35]. 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) [17] 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 will usually use the meta-variable $x$ 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 $\text{Type}_i$. 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 pairs $(e_1, e_2)$ as $\Sigma x : A. B$, first $\text{fst} e$ and second $\text{snd} e$ projections of dependent pairs, and dependent let $\text{let } x = e \text{ in } e'$. For brevity, we omit the type annotation on dependent pairs, as in $(e_1, e_2)$. Note that let-bound definitions do not include type annotations; this is not standard, but type checking is still decidable [35], and it simplifies our ANF translation. For brevity, we omit base types from this formal system but will freely use base types like booleans in examples.

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

---

4We describe in Section 5 how to extend the ANF translation to support annotated let.
In Figure 2, we give the reductions $\Gamma \vdash e \triangleright e'$ for ECC, which are entirely standard. As usual, we extend reduction to conversion by defining $\Gamma \vdash e \triangleright^* e'$ to be the reflexive, transitive, compatible closure of reduction $\triangleright$. 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 eval as the evaluation function for whole-programs using conversion, which we will use in our compiler correctness proof.

In Figure 3 we define definitionally equivalence (or just equivalence) $\Gamma \vdash e \equiv e'$ as conversion up to $\eta$-equivalence. We usually we the notation $e_1 \equiv e_2$ for equivalence, eliding the environment when it is obvious or unnecessary. We also define cumulativity (subtyping) $\Gamma \vdash A \leq B$, 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], and let, [Let] 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: ANF ECC

Our target language, ECC$^A$, is an ANF-restricted subset of ECC. We continue to use the same typing and conversion rules as ECC, which are permitted to break ANF when computing term equivalence during type checking. However, we define an ANF-preserving machine-like semantics for evaluation of program configurations. Note that this means the definitional equivalence is not suitable for equational reasoning about runtime terms (e.g., reasoning about optimizations), without
which do not reduce, computations we can type check but cannot run in the ANF semantics any longer. ANF translation afterwards. Although ECC is restriction of ECC, we type set it in a bold, red, serif font for clarity, and use the shift in fonts to indicate an explicit shift in how we are treating first-class objects in the language, we cannot express control effects—continuations are syntactically enforced ANF syntactically, but is meant to support ANF transformation and optimization with join points.

5 Although ECC is restriction of ECC, we type set it in a bold, red, serif font for clarity, and use the shift in fonts to indicate an explicit shift in how we are treating terms, i.e., as either ANF-restricted terms still suitable for evaluation, or as unrestricted terms that we can type check but cannot run in the ANF semantics any longer.

We give the syntax for ECC in Figure 6. We impose a syntactic distinction between values V which do not reduce, computations N which eliminate values and can be composed using continuations K, and configurations M which intuitively represent whole programs ready to be executed. A continuation K is a program with a hole, and is composed K[N] with a computation N to form a configuration M. For example, (let x = [·] in snd x)[N] = let x = N in snd x. Since continuations are not first-class objects in the language, we cannot express control effects—continuations are syntactically

\[\text{Fig. 5. ECC Typing}\]

\[\text{Fig. 6. ECC Syntax}\]
Compiling Dependent Types Without Continuations

\[
\begin{align*}
K\langle M \rangle &= M \\
K\langle N \rangle &\overset{\text{def}}{=} K[N] \\
K\langle \text{let } x = N' \text{ in } M \rangle &\overset{\text{def}}{=} \text{let } x = N' \text{ in } K\langle M \rangle \\
K\langle [·] \rangle &\overset{\text{def}}{=} K \\
K\langle \text{let } x = [·] \text{ in } M \rangle &\overset{\text{def}}{=} \text{let } x = [·] \text{ in } K\langle M \rangle \\
M[M'//x] &= M \\
M[M'//x] &\overset{\text{def}}{=} (\text{let } x = [·] \text{ in } M)\langle M' \rangle
\end{align*}
\]

Fig. 7. Composition of Configurations

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.

In ANF, all continuations are left associated, so substitution can break ANF. Note that β-reduction takes an ANF configuration \( K[\lambda x : A. M] V \) but would naïvely produce \( K[M[V/x]] \). While the substitution \( M[V/x] \) is well-defined, substituting the resulting term, itself a configuration, into the continuation \( K \) could result in the non-ANF term \( \text{let } x = M \text{ in } M' \). In ANF, continuations cannot be nested.

To ensure reduction preserves ANF, we define composition of a continuation \( K \) and a configuration \( M \), Figure 7, typically called renormalization in the literature [34, 24]. When composing a continuation with a configuration, \( K\langle M \rangle \), we essentially unnest all continuations so they remain left associated.\(^6\) Note that these definitions are simplified by our uniqueness-of-names assumption.

Digression on composition in ANF. In the literature, the composition operation \( K\langle M \rangle \) 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 form, composition 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 \( \text{let} \), as in \( \text{let } y = N \text{ in } M \), 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 \( M \) into a continuation \( \text{let } y = [·] \text{ in } M' \), since this would result in the non-ANF (but valid monadic) expression \( \text{let } y = M \text{ in } M' \). Instead, ANF requires a new operation to compose configurations: \( K\langle M \rangle \). This operation is more generally known as hereditary substitution [37], a form of substitution that maintains canonical forms. So we can think of it as a form of substitution, or, simply, as composition.

We present the call-by-value (CBV) evaluation semantics for ECC\(^A\) in Figure 8. It is essentially standard, but recall that β-reduction produces a configuration \( M \) which must be composed with the

\(^6\)Some work uses an append notation, e.g., \( M :: K \) [34], suggesting we are appending \( K \) onto the stack for \( M \); we prefer notation that evokes composition.
existing continuation \( K \). This semantics is only for the evaluation of configurations; during type checking, we continue to use the type system and conversion relation defined in Section 3.

### 4.1 The Essence of Dependent Continuation Typing

We define continuation typing in Figure 9. The type \((N : A) \Rightarrow B\) of a continuation expresses that this continuation expects to be composed with a term equal (syntactically) to the computation \(N\) of type \(A\) and returns a result of type \(B\) when completed. This is the formal statement that \(N\) is depended upon (in the sense introduced in Section 2) in the rest of the computation, and is key to recovering the dependency disrupted during ANF translation. For the empty continuation \([\cdot]\), \(N\) 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 \(K[N]\) 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 exactly which term \(N\) will be bound in the hole. We see this formally from the typing rule \([\text{Let}]\) (the same for ECC\(^A\) as for ECC in Section 3), since showing that \(\text{let } y = N \text{ in } M\) is well-typed requires showing that \(y = N \vdash M : B\), that is, requires knowing the definition \(y = N\). If we 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 4.4 (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 \(K : (N : A) \Rightarrow B\), \(B\) does not depend on \(N\). This is not important for our work, but is interesting as it provides insight into related work as we discuss in Section 7. The restriction is not necessary, and is not true in all systems, but turns out to be true in ANF. 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 \(K : (N : A) \Rightarrow B\) 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 $\bot$.

### 4.2 Meta-Theory

Since ECC$^A$ is merely a syntactic discipline in ECC, we inherit most of the meta-theory from ECC, notably: logical consistency, type safety, and decidability [27, 35]. There are some new meta-theoretic questions to answer, though, such as: Is the ANF evaluation semantics sound? Does continuation typing make sense?

First, we prove that our ANF evaluation semantics is sound with respect to definitional equivalence. That is, running in our ANF evaluation semantics produces an equivalent value to normalization in the equivalence relation. The heart of this proof is actually naturality, a property found in the literature on continuations that essentially expressed freedom from control effects.

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. 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 $S T(K[M])$. The boundary indicates the term is a target ($T$) term on the inside but a source ($S$) term on the outside. The boundary is only meant to communicate with the reader that a term is no longer in ANF; it has no meaning operationally.

First, we prove that composing continuations in ANF is sound with respect to substitution. This is an expression 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 $K$.

**Lemma 4.1 (Naturality).** $K \langle\langle M \rangle\rangle \equiv ST(K[M])$

**Proof.** By induction on the structure of $M$

- **Case:** $M = N$ trivial
- **Case:** $M = \text{let } x = N' \text{ in } M'$
  
  Must show that
  
  $\text{let } x = N' \text{ in } K \langle\langle M' \rangle\rangle \equiv ST(K[\text{let } x = N' \text{ in } M])$.

  This requires breaking ANF while computing equivalence.

  $\text{let } x = N' \text{ in } K \langle\langle M' \rangle\rangle \triangleright_{\tilde{\xi}} ST(K \langle\langle M' \rangle\rangle)[N'/x]$

  note: this substitution is undefined in ANF

  $= K \langle\langle ST(M'[N'/x]) \rangle\rangle$

  by uniqueness of names

  $\triangleleft^* ST(K[\text{let } x = N' \text{ in } M])$

  by $\tilde{\xi}$-reduction and congruence

  $\square$

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 4.2 (Small-step soundness).** If $M \rightarrow M'$ then $M \equiv M'$
The exported definition represents all but the last machine steps that will happen when executing $M$. We define $\text{defs}[M]$ 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 $K(M)$. Note that $\text{hole}[M]$ will only be well typed in the environment for $M$ extended with the definitions $\text{defs}[M]$.

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 $\text{defs}[M]$, $\text{hole}[M] \equiv M$. In essence, this lemma shows how ANF converts a dependency on a configuration $M$ into a series of dependencies on values,
\[ [e] \mathbf{K} = M \]

\[
\begin{align*}
\{e\} & \overset{\text{def}}{=} [e] [\cdot] \\
[x] \mathbf{K} & \overset{\text{def}}{=} \mathbf{K}[x] \\
[\text{Prop}] \mathbf{K} & \overset{\text{def}}{=} \mathbf{K}[\text{Prop}] \\
[\text{Type}_i] \mathbf{K} & \overset{\text{def}}{=} \mathbf{K}[\text{Type}_i] \\
[\Pi x : A. B] \mathbf{K} & \overset{\text{def}}{=} \mathbf{K}[\Pi x : [A]. [B]] \\
[\lambda x : A. e] \mathbf{K} & \overset{\text{def}}{=} [\lambda x : [A]. [e]] \\
[e_1 \ e_2] \mathbf{K} & \overset{\text{def}}{=} \langle [e_1] \ \text{let} \ x_1 = [\cdot] \ \text{in} \ e_2 \rangle \ \text{let} \ x_2 = [\cdot] \ \text{in} \ [K[x_1 \ x_2]] \\
[\Sigma x : A. B] \mathbf{K} & \overset{\text{def}}{=} \mathbf{K}[\Sigma x : [A]. [B]] \\
[\langle e_1, e_2 \rangle \text{as } A] \mathbf{K} & \overset{\text{def}}{=} \langle [e_1] \ \text{let} \ x_1 = [\cdot] \ \text{in} \ e_2 \rangle (\text{let} \ x_1 = [\cdot] \ \text{in} \ K[(\langle x_1, x_2 \rangle \text{as } [A]))] \\
[\text{fst } e] \mathbf{K} & \overset{\text{def}}{=} [e] \ \text{let} \ x = [\cdot] \ \text{in} \ [K[\text{fst } x]] \\
[\text{snd } e] \mathbf{K} & \overset{\text{def}}{=} [e] \ \text{let} \ x = [\cdot] \ \text{in} \ [K[\text{snd } x]] \\
[\text{let } x = e \ \text{in } e'] \mathbf{K} & \overset{\text{def}}{=} [e] \ \text{let} \ x = [\cdot] \ \text{in} \ [e'] \ K
\end{align*}
\]

Fig. 11. ANF Translation

i.e., the names \(x_0, \ldots, x_{n+1}\) in \(\text{defs } [M]\). Note that the ANF guarantees that all dependent typing rules, like \(\lor \lor' : B[V'/x]\), only depend on values. This lemma allows us to recover the dependency on a configuration.

**Lemma 4.5.** \(\text{defs } [M] \vdash \text{hole } [M] \equiv M\)

**Proof.** Note that the exports \(\text{defs } [M]\) are exactly the definitions from the syntax of \(M\). Inlining those definitions via \(\delta\)-reduction is the same as reducing \(M\) via \(\zeta\)-reduction.

\[
M = (\text{let } x_1 = N_1 \ \text{in} \ \ldots \ \text{let } x_n = N_n \ \text{in} \ N_{n+1})
\]

(6)

\[
\triangleright^\delta \ N_{n+1}[N_1 \ldots N_n/x_1 \ldots x_n]
\]

(7)

And \(\text{hole } [M] = N_{n+1} \triangleright^\delta N_{n+1}[N_1 \ldots N_n/x_1 \ldots x_n]\)

\[\Box\]

5 ANF TRANSLATION

The ANF translation is presented in Figure 11. The translation is defined inductively on the syntax of the source term and is indexed by a current continuation. The translation is essentially standard. When translating a value such as \(x, \lambda x : A. e, \text{ and } \text{Type}_i\), 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 for input to a continuation, then defining the translation and typing preservation proof are somewhat more complicated. For instance, if we required the \text{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. We discuss the implications of this in Section 7.

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 [7, 8, 10, 13, 12]. Since type checking requires definitional equivalence, in the \([\text{Conv}]\) rule, and substitution, in rules such as \([\text{App}]\), 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 the anonymous supplementary technical appendix.

We stage the type preservation proof as follows. First, we show \textit{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 \textit{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 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 \textit{target language font} to the \textit{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 \(ST()\).

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).** \(\llbracket e \rrbracket K' = \text{let } x_1 = N_1 \text{ in } \ldots \text{let } x_n = N_n \text{ in } K'[N_{n+1}]\)

As discussed in Section 4, composition in ANF is somewhat non-standard. Normally, we compose via substitution, so the compositionality property we want is \(\llbracket e[e'/x] \rrbracket \equiv \llbracket e \rrbracket [[e']/x]\), 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 \(e\) under continuation \(K\) and then compose it with a continuation \(K'\), or we can first compose the continuations \(K\) and \(K'\) and then translate \(e\) under the composed continuation. Note that this proof is entirely within ECC\(^A\); there are no language boundaries.

**Lemma 5.2 (Compositionality).** \(K'\langle\llbracket e \rrbracket K\rangle = \llbracket e \rrbracket K'\langle K\rangle\)

Next we show compositionality of the translation with respect to substitution (Lemma 5.3). 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 5.3 (Substitution).** \([e[e'/x]] K \equiv ST(([[e] K][[e']/x]))\)

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 \(\zeta\)-reductions.

**Lemma 5.4.** \(\text{If } \Gamma \vdash e \triangleright e' \text{ then } [[\Gamma]] \vdash [[e]] \equiv [[e']]\).

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 [13, 12], although it does not invalidate their theorems. The past proofs only account for transitivity of \(\triangleright^*\), 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 5.5.** \(\text{If } \Gamma \vdash e \triangleright e' \text{ then } [[\Gamma]] \vdash [[e]] \equiv [[e']]\)

**Proof.** By induction on the structure of \(\Gamma \vdash e \triangleright e'\).

**Case:** \([\text{Red-Refl}]\), trivial.

**Case:** \([\text{Red-Trans}]\), by Lemma 5.4 and the induction hypothesis.

**Case:** \([\text{Red-Cong-Let}]\)

We have \(\Gamma \vdash \text{let } x = e_1 \text{ in } e \triangleright \ast \text{ let } x = e_1 \text{ in } e'\) and \(\Gamma \vdash e \triangleright^* e'\).

We must show that \([[\Gamma]] \vdash [[\text{let } x = e_1 \text{ in } e]] \equiv [[\text{let } x = e_1 \text{ in } e']]\).

\[
[[\text{let } x = e_1 \text{ in } e]] = [[\text{let } x = e_1 \text{ in } y[e/y]]] \equiv ST([[\text{let } x = e_1 \text{ in } y][[e]/y]]) \quad \text{by Lemma 5.3 (Substitution)}
\]

\[
\equiv ST([[\text{let } x = e_1 \text{ in } y][[e']/y]]) \quad \text{by the induction hypothesis applied to } e \triangleright^* e'
\]

\[
\equiv [[\text{let } x = e_1 \text{ in } y[e'/y]]] \quad \text{by Lemma 5.3}
\]

\[
\equiv [[\text{let } x = e_1 \text{ in } e']]
\]

\(\Box\)

The previous two lemmas imply equivalence preservation. Including \(\eta\)-equivalence makes this non-trivial, but not hard.

**Lemma 5.6.** \(\text{If } \Gamma \vdash e \equiv e' \text{ then } [[\Gamma]] \vdash [[e]] \equiv [[e']]\)

Since we implement cumulative universes through subtyping, we must also show subtyping is preserved (Lemma 5.7). The proof is completely uninteresting, except insofar as it is simple, while it seems to be impossible for CPS translation [13]. We discuss this further in Section 7.

**Lemma 5.7.** \(\text{If } \Gamma \vdash e \preceq e' \text{ then } [[\Gamma]] \vdash [[e]] \preceq [[e']]\)

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 \(\text{defs}_e\), expects the term \(\text{hole}_{e}\), 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 5.10 (Type Preservation), but we need a stronger lemma first (Lemma 5.9). 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 4.4 (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 5.8) 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 type preservation.

**Lemma 5.8.** If for all $\Gamma \vdash e : A$ and $[\Gamma], \text{defs}[e] + K : (\text{hole}[e] : [A]) \Rightarrow B$ we know that $[\Gamma] + [e] K : B$, then $[\Gamma], \text{defs}[e] \vdash \text{hole}[e] : [A]$ (and, incidentally, $[\Gamma] + [e] : [A]$)

**Lemma 5.9.**
(1) If $\Gamma \vdash e$ then $[\Gamma]$
(2) If $\Gamma \vdash e : A$, and $[\Gamma], \text{defs}[e] + K : (\text{hole}[e] : [A]) \Rightarrow B$ then $[\Gamma] \vdash [e] K : B$

**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, $[\text{Snd}]$, $[\text{App}]$, and $[\text{Let}]$. We give a couple representative cases; the other cases are essentially similar.

**Case:** $[\text{Ax-Prop}]$
We must show that $[\Gamma] \vdash [\text{Prop}] K : B$.
By definition of the translation, it suffices to show that $[\Gamma] \vdash K[\text{Prop}] : B$.
Note that $\text{defs}[\text{Prop}] = \cdot$; this property holds for all values.
By Lemma 4.4 (Cut), it suffices to show that
- (a) $\text{hole}[\text{Prop}] = \text{Prop}$, which is true by definition of the translation, and
- (b) $\text{Prop} : [\text{Type}_1]$, which is true by $[\text{Ax-Prop}]$, since $[\text{Type}_1] = \text{Type}_1$.

**Case:** $[\text{Snd}]$
We must show $[\Gamma] \vdash [\text{snd} e] K : B$, where $[\Gamma], \text{defs}[\text{snd} e] + K : (\text{hole}[\text{snd} e] : [B'[\text{fst} e / x]]) \Rightarrow B$ and $\Gamma + e : \Sigma x : A'. B'$.
That is, by definition of the translation, we must show, $[\Gamma] \vdash [e] \text{let} x' = [\cdot] \text{in} \; K[\text{snd} x'] : B$.
Let $K' = \text{let} x' = [\cdot] \text{in} \; K[\text{snd} x']$.
Note that we know nothing further about the structure of the term we’re trying to type check, $[e] K'$. 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 $K'$ (that is, we build up definitions in our accumulator) and then appeal to the induction hypothesis for $e$ with $K'$. 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 $\Gamma + e : \Sigma x : A'. B'$ with $K'$, it suffices to show that: $[\Gamma], \text{defs}[e] + \text{let} x' = [\cdot] \text{in} \; K[\text{snd} x'] : (\text{hole}[e] : [\Sigma x : A'. B']) \Rightarrow B$
By $[\text{K-Bind}]$, it suffices to show that
- (a) $[\Gamma], \text{defs}[e] + \text{hole}[e] : [\Sigma x : A'. B']$, which follows by Lemma 5.8 applied to the induction hypothesis for $\Gamma + e : \Sigma x : A'. B'$.
\[ \Gamma \vdash y \]

true \approx true \quad false \approx false

\[
\begin{array}{c}
\vdash \emptyset \\
\Gamma, x = e \vdash y [x \mapsto y(e)] \\
\Gamma, x : A \vdash y [x \mapsto e]
\end{array}
\]

(b) \([\Gamma], \text{defs}[e] \cdot x' = \text{hole}[e] \vdash K[\text{snd} x'] : B.\]

To complete this case of the proof, it suffices to show Item (b). Note that \(\text{defs}[\text{snd} e] = (\text{defs}[e], x' = \text{hole}[e])\) and \(\text{hole}[\text{snd} e] = \text{snd} x'\).

So, by Lemma 4.4 (Cut), given the type of \(K\), it suffices to show that \([\Gamma], \text{defs}[e], x' = \text{hole}[e] \vdash \text{snd} x' : [B'[\text{fst} e/x]].\)

By Lemma 5.3, \([B'[\text{fst} e/x]] = [B'[\text{fst} e]/x].\)

By [Conv], it suffices to show that \([\Gamma], \text{defs}[e], x' = \text{hole}[e] \vdash \text{snd} x' : [B'[\text{fst} e/x]].\)

Note that we cannot show this by the typing rule [Snd], since the substitution \([B'[\text{fst} e/x]]\) copies an apparently arbitrary expression \([\text{fst} e]\) into the type, instead of the expected sub-expression \(\text{fst} x'\). That is, [Snd] tells us \(\text{snd} x' : [B'[\text{fst} x'/x]]\) but we must show \(\text{snd} x' : [B'[\text{fst} e]/x].\) 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 accumulated in our continuation typing save us. We can show that \(\text{fst} e \equiv \text{fst} x'\), under the definitions we have accumulated from continuation typing. This follows by Lemma 4.5.

Therefore, by [Conv], it suffices to show that \([\Gamma], \text{defs}[e], x' = \text{hole}[e] \vdash \text{snd} x' : [B'[\text{fst} x'/x]].\)

By [Snd], it suffices to show \([\Gamma], \text{defs}[e], x' = \text{hole}[e] + x' : \Sigma x : [A'] B'\), which follows since, as we showed in Item (a), \([e] : \Sigma x : [A'] B'\).

\[\square\]

**Theorem 5.10 (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 add some observable outputs by extending the languages with ground types, such as \text{bool}, whose values are comparable across languages. Without such a relation, the best we can prove is that the translation of the value \(v\) produced in the source is \textit{definitionally equivalent} to the value we get by running the translated term, \(i.e.,\) we would get \([v] \equiv \text{eval}([e])\). 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 12.

We define linking as substitution with well-typed closed terms, and define a closing substitution \(y\) with respect to the environment \(\Gamma\) (also in Figure 12). Linking is defined by closing a term \(e\) such that \(\Gamma \vdash e : A\) with a substitution \(\Gamma \vdash y\), written \(y(e)\). Any \(y\) is valid for \(\Gamma\) if it maps each \(x : A \in \Gamma\) to a closed term \(e\) of type \(A\). For definitions in \(\Gamma\), we require that if \(x = e \in \Gamma\), then \(y[x \mapsto y(e)]\), 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.
6 MACHINE STEPS: A GENERAL INTUITION

The ANF translation has helped us identify machine steps as an intuition because the syntax of ANF reflects exactly the steps of computation in the abstract machine, which correspond exactly to the equalities needed to re-establish dependencies. However, the same idea has been hiding in essentially all related work on type preservation and on combining dependent types with effects. Such work has either added a mechanism for recording and reasoning about machine steps in typing derivations, or failed to and had some key limitation. Below, we explicitly describe the related areas in terms of machine steps and our own work. This explicit comparison may be useful in future for designing a general formalization of machine steps for use in modeling dependently typed low-level or effectful languages.

6.1 In Double Negation CPS

Explicit encodings of machine steps are missing from the double negation CPS translations of Barthe et al. [7] and Barthe and Uustalu [8]. Without machine steps, the translations can’t be type preserving, or at least not decidable so.

Barthe et al. [7] manage to prove type preservation by allowing undecidable type checking in the target language. The problem is that, without machine steps, the translation cannot produce a type annotation for a computation, similar to the problem with the type of computations produced by the ANF translation seen in Section 2. The naïve double-negation CPS translation\(^7\) of our example from Section 2, \(e_1 \, e_2 : B[e_2/x]\), is below.

\[
\begin{align*}
\lambda \, k : [B][[e_2]/x] & \to \bot. \\
[e_1] \, (\lambda \, f : \Pi x : [A], [B]. [e_2] \, (\lambda \, x' : [A], f \, x' \, k))
\end{align*}
\]

What is key is that \(k\) expects an argument of type \([B][[e_2]/x]\), but receives an argument of type \([B][x'/x]\], since by the time \(k\) is called, we’ve forgotten that \(x'\) will take on the value of \([e_2]\). To type check it, we need the translation to somehow record the machine step \(x' = \text{value-of}[[e_2]]\) in the application of \([e_2]\) to its continuation. Without a way to thread machine steps through the typing derivation, this translation cannot be type preserving.

Barthe et al. [7] avoid this problem by never generating annotations for functions. Instead, their translation produces the following.

\[
\begin{align*}
\lambda \, k. \, [e_1] \, (\lambda \, f. \, [e_2] \, (\lambda \, x' : f \, x' \, k))
\end{align*}
\]

We cannot decidably type check this, but an oracle could find exactly the right value that the computation \([e_2]\) would produce and include it in the typing derivation.

Unfortunately, Barthe and Uustalu [8] show that this trick doesn’t scale to \(\Sigma\) types or dependent case analysis. And worse, with dependent case analysis, they show any type-preserving double

\(^7\)In fact, this translation lies to the reader in order to better explain the problem. The real translation requires separate computation and value translation, whose details are not important here.
negation translation would rely on an inconsistent target language. The proof relies on the ability to interpret a machine step in CPS, although they do not phrase the problem this way. In our terms, the problem is that if we can record the machine step \( x' = \text{value-of}[e_2] \), whatever “value-of” is must cast \([e_2]: ([A] \to \bot) \to \bot \to [A] \), i.e., it must use a classical operator, and is inconsistent with dependent type case analysis.

### 6.2 In Parametric CPS

Bowman et al. [13] use a modified CPS translation to allow interpreting machine steps without admitting classical operators by changing the types of computations. Instead of double negation, they use a locally polymorphic answer type. This rules out control effects, so “value-of” does not rely on classical reasoning, and yields an interpretation of machine steps consistent with dependent type theory.

We can see their machine steps in their typing rule \([\text{T-Cont}]\), which we reproduce below.

\[
\Gamma \vdash e : \Pi \; \alpha : \text{Prop}. (A \to \alpha) \to \alpha \quad \Gamma \vdash B : \text{Prop} \quad \Gamma, y = e \; A \; \text{id} \vdash e' : B \quad \text{[T-Cont]}
\]

Note that in CPS, the term \( e' \) from the continuation represents the rest of the program, exactly as \( M \) does in ANF. The rule \([\text{T-Cont}]\) introduces the definition \( y = e \; A \; \text{id} \) when type checking the body of the continuation \( e' \), where \( e \; A \; \text{id} \) interprets the CPS translated term \( e \) as a value by applying \( e \) to the “halt” continuation encoded as the identity function.

The definition \( y = e \; A \; \text{id} \) is an encoding of a machine step. It records the fact that, by the time the rest of the program \( e' \) executes, the machine will have evaluated the computation \( e \) and bound the result in \( y \). This encoding records that the CPS computation \( e \), when run with the halt continuation \( \text{id} \), will run to a value.

Unfortunately, CPS requires a complex encoding of machine steps, and additional axioms to justify that encoding. This encoding only works if we can change the answer type of a computation (requiring answer type polymorphism, and impredicativity to type check this encoding), and if the computation cannot exhibit control effects (requiring linearity in the continuation, enforced in their work by parametricity in the answer type and a lack of effects).

Instead, we use the syntactic structure of \text{let} to give us the analogous definition \( y = N \) in \( M \) for free, without relying on parametricity or impredicativity. This is expressed in our rule \([\text{K-Bind}]\).

Notice in the \([\text{T-Cont}]\) rule above, as in \([\text{K-Bind}]\), the result type \( B \) cannot depend upon \( y \), since the \( B \) is determined outside the scope of the value of \( y \). The same is true when typing continuations in other dependent type systems as well [15], although Miquey [29] gives a CPS translation in which the answer type of certain continuations can depend on their value.

### 6.3 In Dependent Sequent Calculus

Machine steps describe a similar solution given by Miquey [29], to a similar problem in the dependent sequent calculus. The key typing rule in their system is \([\text{Cut}]\), given below.

\[
\Gamma \vdash p : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta; \sigma \vdash \{ \cdot \mid p \} \quad B \in A_\sigma \quad (p \parallel e) : \Gamma \vdash \Delta \quad \text{[Cut]}
\]

The dependent sequent calculus makes continuations object-level terms, \( e \). This rule types checks the \text{command} \( (p \parallel e) \), which executes the continuation \( e \) with the value (or, proof) \( p \). In this rule, \( \sigma \) is a list of equalities to which the rule adds \( \{ \cdot \mid p \} \), which expresses that the continuation \( e \) is well-typed only when computation variable \( \cdot \) takes on the value \( p \). That is, \( \sigma \) is the list of machine steps that have been executed so far. When the command \( (p \parallel e) \) executes, \( e \) will execute after an addition machine step: setting the computation variable \( \cdot \) to \( p \). For this encoding, computations
are not permitted to depend on arbitrary expressions, and instead only allowed to depend on values \( p \). This is necessary to avoid control effects in dependent types. This makes it unclear how to translate all of Coq to sequent calculus as an intermediate language.

Notice also that the type of the stack \( B \) cannot depend on the equality \( \cdot \mid p \). The premise \( B \in A_{\sigma} \) essentially means that \( B \) is equal to \( A \), up to machine steps expressed in \( \sigma \) (before \( \sigma \) is extended with \( \cdot \mid p \)). Since \( A \) is the type of \( p \), and \( A \equiv B \), clearly the type of the stack \( e \) cannot depend on the value in its hole, just as in \([K\text{-BIND}]\).

### 6.4 In Dependent Call-By-Push-Value

Call-by-push-value (CBPV) is similar to our ANF target language, and to CPS target languages. In essence, CBPV is a \( \lambda \)-calculus in monadic normal form suitable for reasoning about call-by-value (CBV) or call-by-name (CBN), due to explicit sequencing of computations \([26]\). It has values, computations, and continuations, as we do, and has compositional typing rules (which inspired much of our own presentation). The structure of CBPV is useful for modeling general effects; all computations should be considered to carry an arbitrary effect, while values do not.

Work on designing a dependent call-by-push-value (dCBPV) runs into some of the same design issues that we see in ANF \([1, 36]\), but critically, avoids the central difficulties introduced in Section 2. However, in avoiding the difficulty, it either fails to yield models of effects such as control effects, or cannot be used as a target language from type theory.

In CBPV, and monadic form generally, there is no distinction between computation and configurations, and \textbf{let} is free to compose configurations. The monadic translation of \( e_1 \ e_2 \), which is problematic in ANF as we saw in Section 2, is given below and is easily type preserving in ECC.

\[
[e_1 \ e_2 : B[e_2/x]] = \text{let } x_1 = [e_1] \ \text{in} \ \text{let } x_2 = [e_2] \ \text{in} \ x_1 \ x_2 : [\llbracket B \rrbracket][\llbracket e_2 \rrbracket]/x
\]

Note that since \textbf{let} can bind the “configurations” \([e_1]\) and \([e_2]\), the typing rule \([\text{LET}]\) and the compositionality lemma suffice to show type preservation, without any reasoning about machine steps. For monadic form; we only need a dependent result type for \textbf{let}.

The dependent typing rule for \textbf{let} without definitions is essentially the rule given by Vákár \([36]\), called the dependent Kleisli extension, to support the CBV monadic translation of type theory into dCBPV, and the CBN translation with strong dependent pairs. Vákár \([36]\) observes that without the dependent Kleisli extension, CBV translation is ill-defined (not type preserving), and CBN only works for dependent elimination of positive types (\textit{i.e.}, for the negative-elimination free (NEF) fragment, discussed further in Section 7). This is the same as the observation made independently by Bowman et al. \([13]\) that type-preserving CBV CPS fails for \( \Pi \) types, in addition to the well-known result that the CBN translation failed for \( \Sigma \) types \([8]\).

The essential typing rules from dCBPV are given below.

\[
\begin{align*}
\Gamma \vdash V : A & \quad \Gamma; \Delta \vdash K : \Pi_{F(x:A)}B \\
\Gamma; \Delta \vdash V'K : B[V/x] & \\
\Gamma, z : UA, \Gamma' \vdash B : \text{ctyp} \quad \Gamma; \cdot \vdash M : FA \quad \Gamma, x : A, \Gamma'[tr x/z] ; \vdash N : B[tr x/z] & \\
\Gamma, \Gamma'[\text{thunk } M/z] ; \vdash M \to x \in N : B[\text{thunk } M/z]
\end{align*}
\]

The first rule composes a continuation \( K \) with a value \( V \), but note that the result type \( B \) can depend on the value \( V \). The second rule gives the dependent Kleisli extension. This binds the result of the configuration \( M \) to \( x \) and runs the configuration \( N \). Note that the result type depends on \( M \). Ignoring the details of thunk, \( U \), and \( F \), which have to do with modeling effects, this is essentially the typing rule for dependent let. This is essentially recording the dependency \( x = M \) in the result type, the usual way dependencies are communicated to the context in a dependently typed source language.
This rule is sufficient in monadic form translation into CBPV, but it is not sufficient to support ANF, since ANF does not allow let binding a configuration. In ANF, the configuration $M$ would need to be decomposed into its computation and its continuation, and the continuation sequenced before this $\text{bind}$. ANF transformation would require a stronger typing rule that records machine steps in the context, as we have done in this work.

Without a notion of machine steps, dCBPV has some key limitations. Vákár [36] points out that, although the dependent Klesli extension supports a CBV translation into dCBPV, there are few models of dCBPV that support effects. In particularly, they show dCBPV cannot support control effects. In our terms, this seems to be because the dependent Klesli extension copies the configuration $M$ into the type, rather than recording the machine step exactly once in the derivation. By copying the configuration, an effect in $M$ could occur twice in two different contexts—one in the term context and once in the type context—which would not properly model the control effect, or worse, result in inconsistency as $M$ could produce two different results in those different contexts. We conjecture that extension of dCBPV with a notion of machine steps in derivations would enable properly modeling effects.

Ahman [1] describes eMLTT, an independently discovered variant of dependent CBPV. eMLTT avoids dependent let altogether, but comes up with many useful models of dependent types with effects. Ahman [1], however, does not give a translation from type theory into eMLTT, so we do not know if we can use eMLTT as a compiler intermediate language and it seems likely that an extension such as the dependent Klesli extension would be required to do so. Furthermore, the work of Vákár [36] suggests that such an extension would break many of the models of eMLTT with effects.

7 RELATED AND FUTURE WORK

7.1 Comparison to CPS

ANF is usually seen in opposition to CPS, so we briefly discuss similarities and differences between our type-preserving ANF and prior work on type-preserving CPS. ANF is favored as a compiler intermediate representation, although not universally. Maurer et al. [28] 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 [24] argues the opposite—that CPS is preferred to ANF—and summarizes the arguments for and against.

The CPS translation of Bowman et al. [13] seeks to develop a type-preserving translation. They add a rule similar to our [K-Bind] to the target language, in order to recover disrupted dependencies. Unfortunately, their encoding does not scale to higher universes, and relies on interpreting all functions as parametric. Formally, the problem with higher universes shows up in the lemma for preservation of subtyping, Lemma 5.7, which does not hold when extending their CPS translation to a language with higher universes. By contrast, we prove that our ANF translation works with higher universes and, since ECC is a subset of ECC, the ANF translation is orthogonal to parametricity.

There is also a difference in the type preservation proof; the CPS translation must be defined on typing derivations, not syntax, which prevents us from applying the translation to a language with typed equivalence. Barthe et al. [7] point out this problem, noting that trying to define the translation on typing derivations introduces a cycle in the type preservation proof. Bowman et al. [13] manage to break this cycle by defining definitional equivalence on untyped terms, following the approach used in CIC, but this is yet another restriction on how CPS can be used. ANF can be type-preserving even with typed definition equivalence.
Most work on CPS for dependent types is primarily concerned with integrating control effects and dependent types, not with type-preserving compilation. Therefore, these CPS translation necessarily make different design choices than what we describe in Section 1. For example, we want to avoid any restriction on the source language. However, one must necessarily restrict certain dependencies to integrate control effects and dependent types, at least for the effectful parts of the language. For our translation, naturality is important, but it is non-goal for effectful terms, or is a goal only up to a boundary. We briefly describe some of these translations.

To justify soundness of a dependent sequent calculus, Miquey [29] present a CPS translation based on double negation. Sequent calculi make co-values (essentially, continuation) explicit in the language, and thus express control effects. Miquey [29] develop their CPS translation and avoid relying on parametricity and impredicativity. They rely on modified source type system to track equalities when checking co-values that depend on expression, which (as we showed in Section 6) is essentially similar to our continuation typing. They use delimited continuations to enforce naturality at certain boundaries, to constrain the scope of effects. They also rely on the negative-elimination-free (NEF) restriction [23], which disallows proofs that contain certain kinds of computations, such as arbitrary dependent projection from \(\Sigma\) types. If we want to compile existing languages, such as Coq, which have significant code bases, we cannot admit restrictions like the NEF restriction.

Cong and Asai [15] give a CPS translation for a dependently typed calculus with a restricted form of inductive types and dependent pattern matching, and with delimited control effects, i.e., \texttt{shift} and \texttt{reset}. Effectful terms cannot be depended upon, thus maintaining consistency, and all terms are CPS translated. However, for pure terms, their translation builds on that of Bowman et al. [13] and thus uses a similar custom IL relying on parametricity and impredicativity. Cong and Asai [15] also notice a connection between the type rule \(\text{T-Cont}\) and commutative cuts (discussed in Section 7.3), and observe that continuations cannot have a result type that depends on the continuation argument. Our continuation typing helps separate these issues from the details of \texttt{shift} and \texttt{reset} and CPS.

Pédrot [32] uses a non-standard CPS translation to internalize classical reasoning in a modality encoded in the Calculus of Inductive Constructions (CIC), in which dependency is restricted to ensure consistency, but all terms can be CPS translated. Essentially, each type is translated to a dependent pair of its double negation and a new parametric variant of the type. This CPS translation once again relies on parametricity, and the work leaves as an open question whether it is consistent with recent extensions to type theory, in particular, with univalence. Furthermore, since the translation produces pairs that require further interpretation, it is unclear if this can be used to encode low-level control. It is also unclear how this translation preserving subtyping (Lemma 5.7), since CIC includes an infinite hierarchy of universes, and some universes may be impredicative. It may be that the modality and CPS translation are restricted to \texttt{Prop}.

### 7.2 Branching and Join Points

Our work on type-preserving ANF is ultimately aimed at developing a type-preserving compiler for Coq. Our source contains almost all of the core features of dependency, but one remains: dependent case analysis.

It is well-known that ANF in the presence of branching constructs, such as dependent pattern matching or even just \texttt{if}, can cause considerable code duplication for branches. For instance, supposing we have an (for the moment, non-dependent) \texttt{if}, the naïve ANF translation is the following.

\[
\text{\llbracket if } e \text{ then } e_1 \text{ else } e_2 \text{\rrbracket}_K = [e] \text{ let } x = [\cdot] \text{ in if } x \text{ then } (\llbracket e_1 \rrbracket_K) \text{ else } (\llbracket e_2 \rrbracket_K)
\]
The real problem arises if we have dependent case analysis, in which the result type of the branches can depend on the scrutinee of the case analysis. For simplicity, we consider dependent if. With dependent if, we would have the following standard typing rule.

$$\frac{\Gamma, y : \text{bool} \vdash B : U \quad \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : B[true/y] \quad \Gamma \vdash e_2 : B[false/y]}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : B[e/y]}$$

We can describe the problem clearly using continuation typing. The ANF translation of this term is with respect to the continuation $K : (\text{hole}[\text{if } e \text{ then } e_1 \text{ else } e_2] : [B[e/y]]) \Rightarrow B'$. However, the ANF translation will use $K$ in an ill-typed way, producing in one branch $[e_1]K$, and in the other branch $[e_2]K$. For the first branch, for instance, we must show that now $K : (\text{hole} [e_1] : [B[true/y]]) \Rightarrow B'$. The definitions, def $s[e]$, introduced by the translation of if, are not sufficient to show that this type is equivalent to the type of $K$ expected for the translation of if. Bowman et al. [13] point out an essentially similar problem with CPS for dependent case.

We could resolve this if we could assume while type checking the first branch that $e = true$, and similarly for the second branch that $e = false$. These essentially record additional machine steps: that $e$ will evaluate to true (in one branch) and false (in the other). However, making such assumptions could result in an inconsistency if we’re not careful with the encoding. For example, suppose $e$ is the constant true; then, while type checking, we assume true = false in one branch. Since in ECC$^A$ we require reduction under both branches during type checking, reduction under this inconsistent assumption could cause divergence. This is not a problem in CPS, since the reduction will eventually reach a normal form waiting for a continuation from the outer context.

To enable us to make assumptions like the above but ensure strong normalization, we could encode these machine steps using explicit equality proofs and coercions. The coercions would block reduction until they have a proof of equality, and since we would never end up with a proof that true = false, the inconsistent assumption can never be used in reduction. But the coercion would allow the ANF translation to type check.

With this idea, the type rule for if would be the following.

$$\frac{\Gamma, y : \text{bool} \vdash B : U \quad \Gamma \vdash e : \text{bool} \quad \Gamma, p : e = true \vdash e_1 : B[true/y] \quad \Gamma, p : e = false \vdash e_2 : B[false/y]}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : B[e/y]}$$

Then, the naïve ANF translation for dependent if would be the following.

$$[\text{if } e \text{ then } e_1 \text{ else } e_2] K = [e] \text{ let } x = [\cdot] \text{ in } \text{if } x \text{ then } [e_1] \text{ let } x_1 = [\cdot] \text{ in } K[\text{subst}_p x_1]$$

$$\text{else } [e_2] \text{ let } x_2 = [\cdot] \text{ in } K[\text{subst}_p x_2]$$
We would add the identity type \( e_1 = e_2 \) and the term \( \text{subst}_p e \) to eliminate identity type (derivable from only axiom \( J \)), with the following standard semantics. Recall that the result type of \( K \) cannot depend on the term in its hole, so we do not need a similar conversion for the result of \( K \).

\[
\Gamma, x : A \vdash B : U \quad \Gamma \vdash p : e_1 = e_2 \quad \Gamma \vdash e : B[e_1/x] \\
\frac{}{\Gamma \vdash \text{subst}_p e : B[e_2/x]} \quad \frac{}{\Gamma \vdash \text{refl} e : e = e}
\]

The translation with join points is below, and requires no further additions.

\[
\begin{align*}
&\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ } K = [e] \text{ let } x = [\cdot] \text{ in let } j = \lambda x' : A. K[x'] \text{ in } \\
&\quad \text{if } x \text{ then } [e_1] \text{ let } x_1 = [\cdot] \text{ in subst}_p x_1 \\
&\quad \text{else } [e_2] \text{ let } x_2 = [\cdot] \text{ in subst}_p x_2
\end{align*}
\]

Based on recent work, we conjecture that this is type preserving. Cong and Asai [15] use essentially the same approach to extend the CPS translation of Bowman et al. [13] to inductive types with dependent case analysis and prove type preservation. In the branches of a match, they add equivalences \( e \equiv c e' \) ... between the scrutinee \( e \) and the constructor \( c \) applied to its arguments \( c e' \) .... They prove this extension is type preserving for CPS. Their extension is simple to encode using the identity type, and join points are a local use of CPS, so their work strongly suggests the above extensions to our ANF translation are type preserving.

### 7.3 Dependent Pattern Matching and Commutative Cuts

The above problem with dependent if is related to the problem of commutative cuts [11, 22]. Formally, the problem of commutative cuts can be phrased as: Is the following transformation type-preserving?

\[
K[\text{if } e \text{ then } e_1 \text{ else } e_2] \leadsto \text{if } e \text{ then } K[e_1] \text{ else } K[e_2]
\]

ANF necessarily performs this transformation, as shown in the previous section.

Ignoring ANF for a moment, in general this is not type preserving and continuation typing shows why. Suppose that we have a non-ANF continuation \( K \) with the following typing derivation.

\[
\Gamma, y' : \text{bool} \vdash B' : U' \quad \Gamma, y : B'[e/y'] \vdash B : U \\
\frac{}{\Gamma \vdash K : ((\text{if } e \text{ then } e_1 \text{ else } e_2) : B'[e/y']) \Rightarrow B[(\text{if } e \text{ then } e_1 \text{ else } e_2)/y]}
\]

Note that the result type \( B \) of the continuation \( K \) depends on the term in the hole. (Recall that this would not be true if \( K \) were in ANF, as mentioned in Section 4.1.) The problem is that after the commutative cut, we must show that \( K[e_1] \) and \( K[e_2] \) are well-typed in the branches, which is not true in general since \( K \) expects exactly the term \( e \) if \( e \) then \( e_1 \) else \( e_2 \).

If we add the equalities \( e = \text{true} \) and \( e = \text{false} \) while type checking the branches, as proposed in the previous section, then it appears that we can make the terms \( K[e_1] \) and \( K[e_2] \) well typed. For example, in the first branch after the commutative cut, we will have \( K[e_1] \), but \( K \) expects \( e \) then \( e_1 \) else \( e_2 \). However, since \( e = \text{true} \), these are if \( e \) then \( e_1 \) else \( e_2 \equiv e_1 \), so this should be allowed to type check.

Unfortunately, without ANF or CPS restrictions, we still cannot show the commutative cut is type preserving. In the general case, not only can the continuation \( K \) depend on the term in the hole, so can the result type \( B \). And now, the two branches of the if have different types, and types that are not equivalent outside the context of the if statement: \( K[e_1] : B[e_1/y] \), while \( K[e_2] : B[e_2/y] \). Since the branches of an if must have the same type (up to equivalence), it appears that we must show \( e_1 \equiv e_2 \).

In fact, what we need is essentially another, smaller, commutative cut. Viewing \( B \) as a type-level continuation, we must show \( B[\text{if } e \text{ then } e_1 \text{ else } e_2] \equiv \text{if } e \text{ then } B[e_1] \text{ else } B[e_2] \). This is smaller in the sense that the type of this type cannot contain a commutative cut. For booleans, we could pursue this by adding yet another appeal to the axiom \( J \), but this approach does not scale to indexed inductive types.
In general, existing work on commutative cuts for general inductive types seem to require axiom \( K \), which is undesirable since \( K \) is inconsistent with univalence. Boutillier [11] give an extension to CIC that allows typing commutative cuts by relaxing the termination checker. Explaining the solution is out of scope for this work. We only want to point out two things. First, the solution adds an environment of equalities to the termination checker, much like the machine steps we’ve been focusing on as an intuition. Second, the solution requires axiom \( K \), a very strong requirement which is inconsistent with certain extensions to type theory, such as univalence.

This raises the question: does ANF in general require axiom \( K \)?

We think the answer is no, and the reason is the property we’ve been observing since Section 4.1: in ANF (and CPS, and sequent calculus), the result type of a continuation cannot depend on the term in the hole, thus we never need the second appeal to a commutative cut in the type level. Recording machine steps seems to be enough. The additional structure imposed by ANF seems to avoid some problems of commutative cuts, and we are hopeful that it will be enough to scale ANF to indexed inductive types without additional requirements on the type theory.

We have one additional reason to be hopeful: even if we need a stronger axiom than \( J \), work on dependent pattern matching suggests that univalence may replace axiom \( K \). Recent work on dependent pattern matching creates typing rules similar to what we suggest above to yield additional equalities during a pattern match [5, 14]. There is another unfortunate similarity: much work on dependent pattern matching requires axiom \( K \). In particular, Barras et al. [5] give a new eliminator for CIC which adds additional equalities while checking branches of an elimination, and show that this new typing rule is equivalent to axiom \( K \). However, Cockx et al. [14] discuss a proof-relevant view of unification, in the context of Agda’s dependent pattern matching. They note that normally the heterogeneous equalities usually required by dependent pattern matching require axiom \( K \) to be useful. But, they manage to avoid axiom \( K \) by building on an idea from homotopy type theory that one equality can “layer over” another, to get a proof relevant unification algorithm that does not rely on \( K \), and yet yields the additional equalities for dependent pattern matching.

### 7.4 Monadic Normal Form

Monadic normal form has been studied for compilation [9], and our observations about work on CBPV suggest that monadic normal form may be even more well-behaved than ANF for dependent type theory. However, monadic form is in a sense less low-level than ANF. It still allows nested expressions, meaning there is still more work for the compiler to do to reach assembly language. So while type preservation to monadic form may be simpler, type preservation from monadic form to a lower level language may involve some of the same challenges we solve in this work.

### 8 CONCLUSION

We develop a type-preserving ANF translation for ECC—a significant subset of Coq including dependent functions, dependent pairs, 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


