Compiling Dependent Types Without Continuations (Technical Appendix)

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

TECHNICAL APPENDIX
This document includes extended figures, proofs, and discussion for the paper of the same title.

1 SOURCE: ECC WITH DEFINITIONS
Our source language, ECC, is Luo’s Extended Calculus of Constructions (ECC) [8] extended with definitions [11]. 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) [6] 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 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 Π x : A. B, functions λ x : A. e, application e1 e2, dependent pair types Σ x : A. B, dependent pairs (e1,e2) as Σ x : A. B, first fst e and second snd e projections of dependent pairs, and dependent let let x = e in e'. For brevity, we omit the type annotation on dependent pairs, as in (e1,e2). Note that let-bound definitions do not include type annotations; this is not standard, but type checking is still decidable [11], and it simplifies our ANF translation1. 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.

In Figure 2, we give the reductions Γ ⊢ e ⊳ e' for ECC, which are entirely standard. As usual, we extend reduction to conversion by defining Γ ⊢ e ⊳ e' to be the reflexive, transitive, compatible closure of reduction ⊳. The conversion relation, defined in Figure 3, is used to compute equivalence between types, but we can also view it as the operational semantics for the language. We define eval(e) as the evaluation function for whole-programs using conversion, which we will use in our compiler correctness proof.

In Figure 4 we define definitional equivalence (or just equivalence) Γ ⊢ e ≡ e' as conversion up to η-equivalence. We usually we the notation e1 ≡ e2 for equivalence, eliding the environment when it is obvious or unnecessary. We also define cumulativity (subtyping) Γ ⊢ A ⊲ B, to allow types in lower universes to inhabit higher universes.

We define the type system for ECC in Figure 5, which is mutually defined with well-formedness of environments in Figure 6. The typing rules are entirely standard for a dependent type system. Note that types themselves, such as Π 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 [Prop-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.

---

1 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.

1 We describe in Section 3 how to extend the ANF translation to support annotated let.

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

\[ U ::= Prop \mid \text{Type} \]

Expressions

\[ e, A, B ::= x \mid U \mid \Pi x : A. B \mid \lambda x : A. e \mid e \cdot e \mid \Sigma x : A. B \mid e_1, e_2 \text{ as } \Sigma x : A. B \mid \text{fst } e \mid \text{snd } e \mid \text{let } x = e \text{ in } e \]

Environments

\[ \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, x = e \]

---

Fig. 1. ECC Syntax

\[
\begin{align*}
\Gamma + e & \rightarrow e' \\
\text{x } & \rightarrow_{\delta} e \quad \text{where } x = e \in \Gamma \\
(\lambda x : A. e_1) e_2 & \rightarrow_{\beta} e_1[e_2/x] \\
\text{fst } (e_1, e_2) & \rightarrow_{\pi_1} e_1 \\
\text{snd } (e_1, e_2) & \rightarrow_{\pi_2} e_2 \\
\text{let } x = e \text{ in } e' & \rightarrow_{\xi} e'[e/x]
\end{align*}
\]

\[
\Gamma + e \rightarrow^* e' \quad \Gamma, x : e + e_1 \rightarrow^* e_2 \\
\Gamma + \text{let } x = e \text{ in } e_1 \rightarrow^* \text{let } x = e \text{ in } e_2 \quad [\text{Red-Cong-Let}] \\
\Gamma + e \rightarrow^* e \quad [\text{Red-Refl}] \\
\Gamma + e \rightarrow^* e' \quad [\text{Red-Trans}]
\]

\[ \text{eval}(e) = v \]

\[ \text{eval}(e) = v \quad \text{where } e \rightarrow^* v \text{ and } v \not\rightarrow v' \]

---

Fig. 2. ECC Reduction, Conversion, and Evaluation (excerpts)
\[ \Gamma \vdash A \triangleright^* A' \quad \text{[Red-Cong-Lam1]} \]
\[ \Gamma \vdash \lambda x : A. M \triangleright^* \lambda x : A'. M \quad \text{[Red-Cong-Lam2]} \]
\[ \Gamma, x : A \vdash M \triangleright^* M' \quad \text{[Red-Cong-Pi1]} \]
\[ \Gamma, x : A \vdash \Pi x : A. M \triangleright^* \Pi x : A. M' \quad \text{[Red-Cong-Pi2]} \]
\[ \Gamma, x : A \vdash M \triangleright^* M' \quad \text{[Red-Cong-Sig1]} \]
\[ \Gamma, x : A \vdash \Sigma x : A. M \triangleright^* \Sigma x : A. M' \quad \text{[Red-Cong-Sig2]} \]
\[ \Gamma \vdash V_1 \triangleright^* V'_1 \quad \text{[Red-Cong-Pair1]} \]
\[ \Gamma \vdash V_2 \triangleright^* V'_2 \quad \text{[Red-Cong-Pair2]} \]
\[ \Gamma \vdash (V_1, V_2) \text{ as } A \triangleright^* (V'_1, V'_2) \text{ as } A \quad \text{[Red-Cong-Pair3]} \]
\[ \Gamma \vdash V_2 \triangleright^* V'_2 \quad \Gamma \vdash V_1 \triangleright^* V'_1 \quad \text{[Red-Cong-App1]} \]
\[ \Gamma \vdash V \triangleright^* V' \quad \text{[Red-Cong-App2]} \]
\[ \Gamma \vdash \text{fst } V \triangleright^* \text{fst } V' \quad \text{[Red-Cong-Fst]} \]
\[ \Gamma \vdash V \triangleright^* V' \quad \text{[Red-Cong-Snd]} \]
\[ \Gamma \vdash \text{snd } V \triangleright^* \text{snd } V' \quad \text{[Red-Cong-Snd]} \]
\[ \Gamma, x = N \vdash M \triangleright^* M' \quad \text{[Red-Cong-Let]} \]
\[ \Gamma, x = N \vdash M \triangleright^* M' \quad \text{[Red-Cong-Let]} \]

Fig. 3. ECC Congruence Conversion Rules

\[ \Gamma \vdash e \equiv e' \]
\[ \Gamma \vdash e_1 \triangleright^* e \quad \Gamma \vdash e_2 \triangleright^* e \quad \Rightarrow \quad \Gamma \vdash e_1 \equiv e_2 \quad \text{[\equiv]} \]
\[ \Gamma \vdash e_1 \triangleright^* e'_1 \quad \Gamma \vdash e_2 \triangleright^* \lambda x : A. e \quad \Gamma, x : A \vdash e \equiv e'_1 x \quad \Rightarrow \quad \Gamma \vdash e_1 \equiv e_2 \quad \text{[\equiv-\eta_1]} \]
\[ \Gamma \vdash e_1 \triangleright^* e'_1 \quad \Gamma \vdash e_2 \triangleright^* \lambda x : A. e \quad \Gamma, x : A \vdash e'_1 x \equiv e \quad \Rightarrow \quad \Gamma \vdash e_1 \equiv e_2 \quad \text{[\equiv-\eta_2]} \]

\[ \Gamma \vdash A \preceq B \]
\[ \Gamma \vdash A \equiv B \quad \Rightarrow \quad \Gamma \vdash A \preceq B \quad \text{[\leq-\equiv]} \]
\[ \Gamma \vdash A \preceq B \quad \Gamma \vdash A \preceq B \quad \Rightarrow \quad \Gamma \vdash A \preceq B \quad \text{[\leq-\preceq]} \]
\[ \Gamma \vdash A \preceq B \quad \Gamma, x_1 : A_1 \vdash B_1 \preceq B_2[x_1/x_2] \quad \Rightarrow \quad \Gamma \vdash A \preceq B \quad \text{[\leq-\preceq]} \]
\[ \Gamma \vdash A \preceq B \quad \Gamma, x_1 : A_1 \vdash B_1 \preceq B_2[x_1/x_2] \quad \Rightarrow \quad \Gamma \vdash A \preceq B \quad \text{[\leq-\preceq]} \]

Fig. 4. ECC Equivalence and Subtyping
\[\begin{align*}
\Gamma \vdash e : A \\
\frac{\Gamma \vdash \gamma}{\Gamma \vdash \gamma : \text{Prop} : \text{Type}_0} & \quad \text{[Ax-Prop]} \\
\frac{\Gamma \vdash \gamma}{\Gamma \vdash \gamma : \text{Type}_i : \text{Type}_{i+1}} & \quad \text{[Ax-Type]} \\
\frac{\Gamma \vdash x : A \in \Gamma}{\Gamma \vdash x : A} & \quad \text{[Var]} \\
\frac{\Gamma \vdash e : A \quad \Gamma, x : A, x = e \vdash e' : B}{\Gamma \vdash \text{let } x = e \text{ in } e' : B[e/x]} & \quad \text{[Let]} \\
\frac{\Gamma \vdash A : \text{Type}_i \quad \text{,} \quad \Gamma, x : A + B : \text{Type}_i}{\Gamma \vdash \Pi x : A. B : \text{Prop}} & \quad \text{[Prod-Prop]} \\
\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A + B : \text{Type}_i}{\Gamma \vdash \Pi x : A. B : \text{Type}_i} & \quad \text{[Prod-Type]} \\
\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash e' : A'}{\Gamma \vdash e e' : B[e'/x]} & \quad \text{[App]} \\
\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A + B : \text{Type}_i}{\Gamma \vdash \Sigma x : A. B : \text{Type}_i} & \quad \text{[Sig]} \\
\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash (e_1, e_2) \text{ as } \Sigma x : A. B : \Sigma x : A. B} & \quad \text{[Pair]} \\
\frac{\Gamma \vdash e : \Sigma x : A. B}{\Gamma \vdash \text{fst } e : A} & \quad \text{[Fst]} \\
\frac{\Gamma \vdash e : \Sigma x : A. B}{\Gamma \vdash \text{snd } e : B[\text{fst } e/x]} & \quad \text{[Snd]} \\
\frac{\Gamma \vdash e : A \quad \Gamma \vdash B : \text{U} \quad \Gamma \vdash A \preceq B}{\Gamma \vdash e : B} & \quad \text{[Conv]} \\
\vdash \cdot \quad \text{[W-Empty]} & \\
\frac{\Gamma \vdash \cdot}{\Gamma \vdash \cdot : \text{U}} & \quad \text{[W-Assum]} \\
\frac{\Gamma \vdash \cdot}{\Gamma, x : A} & \quad \text{[W-Assum]} \\
\frac{\Gamma \vdash \cdot}{\Gamma, x = e} & \quad \text{[W-Def]} \\
\end{align*}\]

Fig. 5. ECC Typing

Fig. 6. ECC Well-Formed Environments
Universes

\[ U ::= \text{Prop} \mid \text{Type}_i \]

Values

\[ V ::= x \mid U \mid \lambda x : M.M \mid \Pi x : M.M \]
\[ \mid \Sigma x : M.M \mid \langle V, V \rangle \]

Computations

\[ N ::= V \mid V V \mid \text{fst} V \mid \text{snd} V \]

Configurations

\[ M, A, B ::= N \mid \text{let } x = N \text{ in } M \]

Continuations

\[ K ::= [\cdot] \mid \text{let } x = [\cdot] \text{ in } M \]

Fig. 7. ECC Syntax

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

Fig. 8. Composition of Configurations

2 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 ANF translation afterwards.\(^2\) Although ECC\(^A\) 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\(^A\) in Figure 7. 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, \((\text{let } x = [\cdot] \text{ in } \text{snd } x)\langle\langle N \rangle\rangle = \text{let } x = N \text{ in } \text{snd } x\). Since continuations are not first-class objects in the language, 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.

In ANF, all continuations are left associated, so substitution can break ANF. Note that \(\beta\)-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 8, typically called renormalization in the literature [10, 7]. When composing a continuation with a configuration, \(K\langle\langle M \rangle\rangle\),

\(^2\)This ability to break ANF locally to support reasoning is similar to the language \(F_J\) of Maurer et al. [9], which does not enforce ANF syntactically, but is meant to support ANF transformation and optimization with join points.
we essentially unnest all continuations so they remain left associated.\(^3\) Note that these definitions are simplified by our uniqueness-of-names assumption.

**Digression on composition in ANF.** In the literature, the composition operation \(K\langle\langle M\rangle\rangle\) is usually introduced as renormalization, as if the only intuition for why it exists is “well, it happens that ANF is not preserved under \(\beta\)-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 \(\beta\)-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 \texttt{let}, as in \(\texttt{let } y = \texttt{N in M}\), 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 \(\texttt{let } y = \texttt{N in M}\), since this would result in the non-ANF (but valid monadic) expression \(\texttt{let } y = \texttt{M in M}\). Instead, ANF requires a new operation to compose configurations: \(K\langle\langle M\rangle\rangle\). This operation is more generally known as hereditary substitution\(^{12}\), 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 9. It is essentially standard, but recall that \(\beta\)-reduction produces a configuration \(M\) which must be composed with the 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 1.
2.1 The Essence of Dependent Continuation Typing

We define continuation typing in Figure 10. 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 ??) 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 1), since showing that let \(y = N\) in \(M\) is well-typed requires showing that \(y = N + M\), 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 2.4 (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. 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 \(\perp\).

2.2 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 [8, 11]. 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 \(\tilde{S}T(M[M'/x])\) and \(\tilde{S}T(K[M])\). 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 2.1 (Naturality).** \(K(\tilde{S}T(M)) \equiv \tilde{S}T(K[M])\)

**Proof.** By induction on the structure of \(M\)

Case: \(M = N\) trivial

Case: \(M = \text{let } x = N' \text{ in } M'\)

\(^3\)Some work uses an append notation, e.g., \(M :: K\) [10], suggesting we are appending \(K\) onto the stack for \(M\); we prefer notation that evokes composition.
ANF evaluation semantics have no congruence rules.

Then we show soundness of the evaluation function.

To reason inductively about ANF terms, we need to separate a configuration $M$ into its exported definitions $\text{defs} [M]$ and its underlying computation $\text{hole} [M]$, which we define formally in Figure 11. The exported definition represent all but the last machine steps that will happen when executing $M$, while the underlying computation $\text{hole} [M]$ is the final computation whose result will be bound when $M$ is composed with another continuation. We define $\text{defs} [M]$ to be the sequence of definitions bound the ANF term $M$. These are the definitions that will be in scope for a continuation $K$

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.

**Lemma 2.2 (Small-step soundness).** If $M \rightarrow M'$ then $M \equiv M'$

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

**Case:** $n = 0$ By $\text{Red-refl}$ and $\equiv$.

**Case:** $n = i + 1$ Follows by Lemma 2.2 and the induction hypothesis. □

**Theorem 2.3 (Evaluation soundness).** $\vdash \text{eval}(M) \equiv M$

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

**Lemma 2.4 (Cut).** If $\Gamma \vdash K : (N : A) \Rightarrow B$ and $\Gamma \vdash N : A$ then $\Gamma \vdash K[N] : B$.

**Proof.** By cases on $\Gamma \vdash K : (N : A) \Rightarrow B$

**Case:** $\Gamma \vdash [\cdot] : (N : A) \Rightarrow A$, trivial

**Case:** $\Gamma \vdash \text{let } y = [\cdot] \text{ in } M : (N : A) \Rightarrow B$

We must show that $\Gamma \vdash \text{let } y = N \text{ in } M : B$, which follows directly from $\text{[let]}$ since, by the continuation typing derivation, we have that $\Gamma, y = N \vdash M : B$ and $y \not\in \text{fv}(B)$. □
when composed with $M$, i.e., in scope for $K$ in $K\langle\langle M\rangle\rangle$. 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., 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, i.e., the names $x_1, \ldots, x_{n+1}$ in $\text{defs}[M]$. Note that the ANF guarantees that all dependent typing rules, like $\text{V V'} : B[V'/x]$, only depend on values. This lemma allows us to recover the dependency on a configuration.

**Lemma 2.5.** $\text{defs}[M] \vdash \text{hole}[M] \equiv M$

**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.

$$M = (\text{let } x_1 = N_1 \text{ in } \ldots \text{ let } x_n = N_n \text{ in } N_{n+1}) \tag{6}$$

$$\overset{\zeta}{\Rightarrow}^n N_{n+1}[N_1 \ldots N_n/x_1 \ldots x_n] \tag{7}$$

And $\text{hole}[M] = N_{n+1} \overset{\delta}{\Rightarrow}^n N_{n+1}[N_1 \ldots N_n/x_1 \ldots x_n] \quad \Box$

Some presentations of evaluation context typing, in non-dependent settings, use a rule link the following.

$$\Gamma, x : A \vdash E[x] : B$$
$$\Gamma \vdash E : A \Rightarrow B$$

This suggests we could define continuation typing as follows.

$$\Gamma \vdash K[N] : B$$
$$\Gamma \vdash K : (N : A) \Rightarrow B \quad \text{[K-Type]}$$

That is, instead of adding separate rules $\text{[K-Empty]}$ and $\text{[K-Bind]}$, we define a well-typed continuation to be one whose composition with the expect term in the whole is well-typed. Then, Lemma 2.4 (Cut) is definitional rather than admissible. This rule is somewhat surprising; it appears very much like the definition of $\text{[Cut]}$, except the computation $N$ being composed with the continuation comes from its type, and the continuation remains un-composed in what we would consider the output of the rule.

The presentations are equivalent, but it is less clear how $\text{[K-Type]}$ is related to the definitions we wish to focus on. It is exactly the premises of $\text{[K-Bind]}$ that we need to recover type-preservation for ANF, so we choose the presentation with $\text{[K-Bind]}$.

However, the rule $\text{[K-Type]}$ is more general in the sense that the continuation typing does not need and changes as the definition of continuations change.
we are interested in the conversion semantics used for definitional equivalence, not in the machine semantics used to formally state type preservation in terms of the intuitive reason that type preservation should hold: because the

The continuation typing allows us to express the inductive invariant required for ANF. The continuation typing allows us then their translations are definitionally equivalent. Finally, we can show type preservation of the ANF translation, evaluating ANF terms. Then, we show

equivalence preservation

substitution. Next, we show that reduction and conversion are preserved up to equivalence. Note that for this theorem, substituting. This proof is somewhat non-standard for ANF since the notion of composition in ANF is not the usual 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 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

\begin{align*}
\llbracket e \rrbracket K = M
\end{align*}

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

Fig. 12. ANF Translation

3 ANF TRANSLATION

The ANF translation is presented in Figure 12. 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 $x, \lambda x : A. e,$ 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 ??.

Our goal is to prove type preservation: if $e$ is well-typed in the source, then $\llbracket e \rrbracket$ 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 [1–3, 5, 4]. Since type checking requires definitional equivalence, in the [Conv] rule, and substitution, in rules such as [Arp], we must preserve definitional equivalence and substitution. Since definitional equivalence is defined in terms of reduction, we must preserve reduction up to equivalence.

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 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 so we elide it.

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 3.1). The proof is straightforward so we elide it.

Theorem 3.1 (ANF). \([e] K' = \text{let } x_1 = N_1 \text{ in } \ldots \text{let } x_n = N_n \text{ in } K'[N_{n+1}]\)

As discussed in Section 2, composition in ANF is somewhat non-standard. Normally, we compose via substitution, so the compositionality property we want is \([e |x| e'] = [e][x/e']\), 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 3.2) 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 3.2 (Compositionality). \(K' \langle [\llbracket e \rrbracket K] \rangle = [\llbracket e \rrbracket K'] \langle K \rangle\)

Proof. By induction on the structure of \(e\). All value cases are trivial. The cases for non-values are all essentially similar, by definition of composition for continuations or configurations. We give some representative cases.

**Case:** \(e = x\) Must show \(K' \langle [\llbracket K|x\rrbracket] \rangle = [\llbracket K|x\rrbracket K']\), which is trivial.

**Case:** \(e = \exists x : A. B\) Must show that \(K' \langle [\llbracket K|\exists x : [A]. [B]]x\rrbracket \rangle = [\llbracket K|\exists x : [A]. [B]]x\rrbracket \langle K \rangle\), which is trivial. Note that we need not appeal to induction, since the recursive translation does not use the current continuation for values.

**Case:** \(e = e_1 e_2\) Must show that\n
\[
K' \langle \langle [e_1] \text{ let } x_1 = [\cdot] \text{ in } ([e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2]) \rangle \rangle = ([e_1] \text{ let } x_1 = [\cdot] \text{ in } ([e_2] \text{ let } x_2 = [\cdot] \text{ in } K' \langle [\llbracket K \rrbracket K'] \rangle x_1 x_2))
\]

The proof follows essentially from the definition of continuation composition.

\[
\begin{align*}
K' \langle \langle [e_1] \text{ let } x_1 = [\cdot] \text{ in } ([e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2]) \rangle \rangle & = ([e_1] \text{ let } x_1 = [\cdot] \text{ in } ([e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2])) \\
& \text{by the induction hypothesis applied to } e_1 \\
& = ([e_1] \text{ let } x_1 = [\cdot] \text{ in } K' \langle [\llbracket e_2 \rrbracket K] \rangle \langle [\cdot] \text{ in } K[x_1 x_2] \rangle) \\
& \text{by definition of continuation composition} \\
& = ([e_1] \text{ let } x_1 = [\cdot] \text{ in } ([e_2] \text{ let } x_2 = [\cdot] \text{ in } K' \langle [\llbracket K \rrbracket K'] \rangle x_1 x_2)) \\
& \text{by the induction hypothesis applied to } e_2 \\
& = ([e_1] \text{ let } x_1 = [\cdot] \text{ in } ([e_2] \text{ let } x_2 = [\cdot] \text{ in } K' \langle [\llbracket K \rrbracket K'] \rangle x_1 x_2)) \\
& \text{by definition of continuation composition}
\end{align*}
\]

\(\square\)

Next we show compositionality of the translation with respect to substitution (Lemma 3.3). 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 3.3 (Substitution). \([e|e'/x|] K = S\langle T([\llbracket e \rrbracket K][e'/x])\rangle\)
Proof. By induction on the structure of $e$. We give the key cases.

**Case:** $e = x$ Must show that $[[e']K] \equiv ST(((x)K)([e']/x))$

$$ST([x]K([e']/x))$$

$$= ST(K[x][[e']/x])$$

$$= ST(K[[e']])$$

$$\equiv K[[e']]$$ by Lemma 2.1

$$\equiv [e']K$$ by Lemma 3.2

**Case:** $e = \Pi x'. A. B$ Must show that $[[\Pi x'. A. B[e'/x]]]K \equiv ST(((\Pi x'. A. B)[K])([e']/x))$

$$[[\Pi x'. A. B[e'/x]]]K$$

$$= [[\Pi x'. A. B[e'/x]]]K$$

$$= K[[\Pi x'. A[e'/x]], [B[e'/x]]]$$

$$= K[[\Pi x'. A[[e']/x]], ST([[B][[e']/x]])]$$ by the induction hypothesis

$$= ST(K[[\Pi x'. A[e'/x]], ST([[B][[e']/x]])])$$ by definition of substitution

$$= ST(((\Pi x'. A. B)[K])([e']/x))$$ by definition

**Case:** $e = e_1 e_2$ Must show that $[[e_1 e_2[e'/x]]]K \equiv ST(((e_1 e_2)[K])([e']/x))$

$$[[e_1 e_2[e'/x]]]K$$

$$= [[e_1[e'/x] e_2[e'/x]]]K$$

$$= [[e_1[e'/x]]] \text{ let } x_1 = [\cdot] \text{ in } [e_2[e'/x]] \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2]$$ by substitution

$$\equiv [e_1[e'/x]] \text{ let } x_1 = [\cdot] \text{ in } ST([[e_2][[e']/x]]) \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2][[e']/x]$$ by translation

$$\equiv [e_1] \text{ let } x_1 = [\cdot] \text{ in } [e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1 x_2][[e']/x][[e']/x]$$ by IH applied to $e_1$

$$\equiv ST([[e_1][e_2][K]][[e']/x][[e']/x])$$ by IH applied to $e_2$

$$= ST(((e_1 e_2)[K][[e']/x]))$$ by substitution

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 3.4.** If $\Gamma \vdash e \equiv e'$ then $[[\Gamma]] \vdash [e] \equiv [e']$.

Proof. By cases on $\Gamma \vdash e \equiv e'$. We give the key cases.

**Case:** $\Gamma \vdash x \equiv e'$.

We know that $x = e' \in \Gamma$, and by definition $x = [e'] \in [[\Gamma]]$, so the goal follows by definition.

**Case:** $\Gamma \vdash \lambda x : A. e_1 e_2 \equiv e_1 e_2 / e_2 / x$.

We must show $[[\Gamma]] \vdash [[\lambda x : A. e_1 e_2]] \equiv [[e_1 e_2 / e_2 / x]]$

$$[[\lambda x : A. e_1 e_2]]$$

$$= [[\lambda x : A. e_1]] \text{ let } x_1 = [\cdot] \text{ in } [e_2] \text{ let } x_2 = [\cdot] \text{ in } x_1 x_2$$

$$= \text{ let } x_1 = (\lambda x : [A]. [e_1]) \text{ in } [e_2] \text{ let } x_2 = [\cdot] \text{ in } x_1 x_2$$

$$\equiv [\cdot] \text{ in } (\lambda x : [A]. [e_1] x_2 [e_2])$$ by Lemma 3.2

$$\equiv \lambda x : A. e_1 e_2$$

(28)
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 [5, 4], 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 3.5.** If $\Gamma \vdash e \triangleright^* e'$ then $[\Gamma] \vdash [e] \equiv [e']$

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

**Case:** [Red-Refl], trivial.

**Case:** [Red-Trans], by Lemma 3.4 and the induction hypothesis.

**Case:** [Red-Cong-Let]
We have $\Gamma \vdash \text{let } x = e_1 \text{ in } e \triangleright^* \text{ let } x = e_1 \text{ in } e'$ and $\Gamma \vdash e \triangleright^* e'$.
We must show that $[\Gamma] \vdash [\text{let } x = e_1 \text{ in } e] \equiv [\text{let } x = e_1 \text{ in } e']$.

\[
[\text{let } x = e_1 \text{ in } e] \\
= [\text{let } x = e_1 \text{ in } y[e/y]] \\
= ST([\text{let } x = e_1 \text{ in } y][[e]/y]) \\
= ST([\text{let } x = e_1 \text{ in } y][[e']]/y) \\
= [\text{let } x = e_1 \text{ in } y[e'/y]] \\
= [\text{let } x = e_1 \text{ in } e'] \\
\]

by Lemma 3.3 (Substitution) (37)

by the induction hypothesis applied to $e \triangleright^* e'$ (38)

by Lemma 3.3 (39)

$\square$

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

**Lemma 3.6.** If $\Gamma \vdash e \equiv e'$ then $[\Gamma] \vdash [e] \equiv [e']$

**Proof.** By induction on the derivation of $\Gamma \vdash e \equiv e'$.

**Case:** [$\equiv$] follows by Lemma 3.5.

**Case:** [$\equiv \cdot \eta_1$]

By Lemma 3.5, we know $[e] \equiv [\lambda x : A. e_1]$. By transitivity, it suffices to show $[\lambda x : A. e_1] \equiv [e']$.

By $[\equiv \cdot \eta_1]$, since $[\lambda x : A. e_1] = \lambda x : [A]. [e_1]$, it suffices to show that $[e_1] \equiv [e'] x_2$

\[
[e_1] \\
= [e' x_2] \\
= [e'] \text{ let } x_1 = [\cdot] \text{ in } x_1 \ x_2 \\
= (\text{let } x_1 = [\cdot] \text{ in } x_1 \ x_2) [e'] \\
= \text{ let } x_1 = [e'] \text{ in } x_1 \ x_2 \\
\]

by the induction hypothesis (41)

by Lemma 3.2 (43)

by Lemma 2.1 (44)

$\square$
Case: \(\equiv \eta_2\] Essentially similar to the previous case.

Since we implement cumulative universes through subtyping, we must also show subtyping is preserved (Lemma 3.7).

The proof is completely uninteresting, except insofar as it is simple, while it seems to be impossible for CPS translation [5]. We discuss this further in ??.

Lemma 3.7. If \(\Gamma \vdash e \leq e'\) then \([\Gamma] \vdash [e] \leq [e']\)

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

Case: \(\leq \equiv\]. Follows by Lemma 3.6.

Case: \(\leq \text{Trans}\]. Follows the induction hypothesis.

Case: \(\leq \text{Prop}\). Trivial, since \(\text{Prop} = \text{Prop}\) and \([\text{Type}_0] = \text{Type}_0\).

Case: \(\leq \text{Cum}\). Trivial, since \([\text{Type}_i] = \text{Type}_i\) and \([\text{Type}_{i+1}] = \text{Type}_{i+1}\).

Case: \(\leq \text{P1}\).

We must show that \([\Gamma] \vdash \Pi x_1 : A_1, B_1 \leq \Pi x_2 : A_2, B_2\]

By definition of the translation, we must show \([\Gamma] \vdash \Pi x_1 : A_1, B_1 \leq \Pi x_2 : A_2, B_2\).

Note that if we lifted the continuations in type annotations \(A_1\) and \(A_2\) outside the \(\Pi\), as CBPV suggests we should, we would need a new subtyping rule that allows subtyping \text{let} expressions. As it is, we proceed by \(\leq \text{P1}\).

It suffices to show that

(a) \([\Gamma] \vdash [A_1] \equiv [A_2]\], which follows by the induction hypothesis.

(b) \([\Gamma], x_1 : A_1, B_1 \vdash [B_2] \leq [B_2] / [x_1/x_2]\], which follows by the induction hypothesis.

Case: \(\leq \text{StG}\]. Similar to previous case.

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 3.10 (Type Preservation), but we need a stronger lemma first (Lemma 3.9). 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 2.4 (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 3.8) 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 hypothesis implies the type preservation theorem we expect.

Lemma 3.8. If for all \(\Gamma \vdash e : A\) and \([\Gamma], \text{defs}[e] \vdash K : (\text{hole}[e] : [A]) \Rightarrow B\) we know that \([\Gamma] \vdash [e] K : B\), then

\(\Gamma, \text{defs}[e] \vdash \text{hole}[e] : [A]\) (and, incidentally, \([\Gamma] \vdash [e] : [A]\))

Proof. Note that by Theorem 3.1 (ANF) and the definitions of \text{defs}[e] and \text{hole}[e], \([\Gamma], \text{defs}[e] \vdash \text{hole}[e] : [A]\) is a sub-derivation of \([\Gamma] \vdash [e] : [A]\), so it suffices to show that \([\Gamma] \vdash [e] : [A]\). By the premise \([\Gamma] \vdash [e] K : B\), it suffices to show that \([\Gamma] \vdash [a] : [A]\), which is true by [K-EMPTY].

Lemma 3.9.

(1) If \(\Gamma \Gamma \vdash \Gamma\)

(2) If \(\Gamma \vdash e : A\), and \([\Gamma], \text{defs}[e] \vdash K : (\text{hole}[e] : [A]) \Rightarrow B\) 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, [SND], [APP], and [LET]. We give these cases, although they are essentially similar, and a couple of other representative cases, which are uninteresting.
Case: $[\text{Ax-Prop}]$

We must show that $[\Gamma] \vdash [\text{Prop} \mid K : B]$.

By definition of the translation, it suffices to show that $[\Gamma] \vdash K[\text{Prop} ] : B$.

Note that $\text{defs}[\text{Prop}] = :$; this property holds for all values.

By Lemma 2.4 (Cut), it suffices to show that
(a) hole$[\text{Prop}] = \text{Prop} ;$, which is true by definition of the translation, and
(b) $\text{Prop} : [\text{Type}]$, which is true by $[\text{Ax-Prop}]$, since $[\text{Type} ] = \text{Type} 1$.

Case: $[\text{Lam}]$

We must show that $[\Gamma] \vdash [\lambda x : A'. e' ] K : B$.

That is, by definition of the translation, $[\Gamma] \vdash K[\lambda x : [A'], [e'] ] : B$.

Recall that $\text{defs}[\lambda x : A'. e'] = :$, since values export no definitions.

By Lemma 2.4, it suffices to show that $[\Gamma] \vdash \lambda x : [A'], [e'] : [\Pi x : A', B']$.

By definition, $[\Pi x : A', B'] = \Pi x : [A'], [B']$. we must show $[\Gamma] \vdash \lambda x : [A'], [e'] : [\Pi x : [A'], [B']$.

By $[\text{Lam}]$, it suffices to show $[\Gamma], x : [A'] \vdash [e'] \vdash [B']$.

Note that $[\Gamma] \vdash \Gamma : \Gamma \Rightarrow (\Gamma : \Gamma) \Rightarrow [B']$.

So, the goal follows by the induction hypothesis applied to $\Gamma, x : A' \vdash e' : B'$ with $K = [\cdot]$.

Case: $[\text{Snd}]$

We must show $[\Gamma] \vdash [\text{snd e}] K : B$, where $[\Gamma], \text{defs}[\text{snd e}] \vdash K : (\text{hole}[\text{snd e}] : \text{snd} e : \text{B'}[\text{fst} e/x]) \Rightarrow B$ and $\Gamma \vdash e : \Sigma x : A'. B'$.

That is, by definition of the translation, we must show, $[\Gamma] \vdash \text{let } x' = [\cdot ] \text{ in } K[\text{snd x'}] : B$.

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 $\Gamma \vdash e : \Sigma x : A'. B'$ with $K'$, it suffices to show that: $[\Gamma], \text{defs}[e] \vdash \text{let } x' = [\cdot ] \text{ in } K[\text{snd x'}] : B$.

By $[\text{K-Bnd}]$, it suffices to show that
(a) $[\Gamma], \text{defs}[e] \vdash \text{hole}[e] : [\Sigma x : A'. B']$, which follows by Lemma 3.8 applied to the induction hypothesis for $\Gamma \vdash e : \Sigma x : A'. B'$.

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

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

By Lemma 3.3, $[\text{B'}[\text{fst} e/x]] = [\text{B'}][[\text{fst} e]/x]$. By $[\text{Conv}]$, it suffices to show that $[\Gamma], \text{defs}[e], x' = \text{hole}[e] \vdash \text{snd x'} : [\text{B'}][[\text{fst} e]/x]$.

Note that we cannot show this by the typing rule $[\text{Snd}]$, since the substitution $[\text{B'}][[[\text{fst} e]/x]$ copies an apparently arbitrary value $\text{fst} e$ into the type, instead of the expected $\text{sub-expression} \text{fst} x'$. That is, $[\text{Snd}]$ tells us $\text{snd x'} : [B'][[\text{fst} e'/x]$ 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 ???. 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 2.5.

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

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

Case: $[\text{App}]$

We must show that $[\Gamma] \vdash [e_1; e_2] K : B$. 


That is, by definition, \( [\Gamma] \vdash [e_1] \text{ let } x_1 = [\cdot] \text{ in } [e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1, x_2] : B \).

Again, we know nothing about the structure of \([e_1] K')\), so we must proceed inside-out.

By the inductive hypothesis applied to \( \Gamma \vdash e_1 : B'[e_1/x] \), it suffices to show that
\[
[\Gamma], \text{defs}_1[e_1] \vdash \text{let } x_1 = [\cdot] \text{ in } [e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1, x_2] : B
\]

To show this, by \([\text{K-Bind}]\), it suffices to show

(a) \( [\Gamma], \text{defs}_1[e_1] \vdash \text{hole}[e_1] : [\Pi x : A'. B'] \), which follows by Lemma 3.8 applied to the induction hypothesis for \( \Gamma \vdash e_1 : \Pi x : A'. B' \).

(b) \( [\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1] + [e_2] \text{ let } x_2 = [\cdot] \text{ in } K[x_1, x_2] : B \)

By the inductive hypothesis applied to \( \Gamma \vdash e_2 : A' \), it suffices to show that
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2] \vdash \text{let } x_2 = [\cdot] \text{ in } K[x_1, x_2] : B
\]

By \([\text{K-Bind}]\), it suffices to show
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2], x_2 = \text{hole}[e_2] \vdash K[x_1, x_2] : B.
\]

By Lemma 2.4 (\text{Cut}), we must show
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2], x_2 = \text{hole}[e_2] \vdash \text{let } x_1 x_2 : [B'[e_2]/x]]
\]

By Lemma 3.5 and \([\text{Conv}]\), it suffices to show
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2], x_2 = \text{hole}[e_2] \vdash x_1 x_2 : [B'[e_2]/x].
\]

As in the proof case for \([\text{snd}]\), we cannot proceed directly by \([\text{app}]\), since we see a disrupted dependency. This dependent application whose type depends on the argument being the specific value \( e_2 \) now finds the argument \( x_2 \). This is the same issue as type-preservation for call-by-value CPS [5]. But, again, we know by Lemma 2.5 that under these exported definitions, \( e_2 \equiv x_2 \). So by \([\text{Conv}]\), it suffices to show
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2], x_2 = \text{hole}[e_2] \vdash \text{let } x_1 x_2 : [B'[e_2/x].
\]

By \([\text{app}]\) it suffices to show

(a) \( [\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2], x_2 = \text{hole}[e_2] \vdash x_1 : [\Pi x : A', [B']]. \)

(b) \( [\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2], x_2 = \text{hole}[e_2] \vdash x_2 : [A']. \)

which follows by Lemma 3.8 applied to the induction hypothesis for \( \Gamma \vdash e_2 : A' \).

Case: \([\text{let}]\)

We must show that \([\Gamma] \vdash [\text{let } x = e_1 \text{ in } e_2] K : B.\)

That is, by definition, \([\Gamma] \vdash [e_1] \text{ let } x_1 = [\cdot] \text{ in } [e_2] K : B.\)

By the inductive hypothesis applied to \( \Gamma \vdash e_1 : A \), it suffices to show that
\[
[\Gamma], \text{defs}_1[e_1] \vdash \text{let } x_1 = [\cdot] \text{ in } [e_2] K : [\text{hole}[e_1] : [A]] \Rightarrow B.
\]

By \([\text{K-Bind}]\), it suffices to show

(a) \( [\Gamma], \text{defs}_1[e_1] \vdash \text{hole}[e_1] : [[A]] \), which follows by Lemma 3.8 applied to the induction hypothesis for \( \Gamma \vdash e_1 : A \).

(b) \( [\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1] + [e_2] K : B.\)

Item (b) follows from the induction hypothesis applied to \( \Gamma, x = e_1 + e_2 : B' \) with \( K \) (the same well-typed \( K \) that we have from our current premise), if we can show that \( K \) is well typed as follows:
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2] \vdash K : ([\text{hole}[e_2] : [B'[e_1/x_1]]) \Rightarrow B
\]

Currently, we know by our premises that
\[
[\Gamma], \text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2] \vdash K : ([\text{hole}[\text{let } x_1 = e_1 \text{ in } e_2] : [B'[e_1/x_1]]) \Rightarrow B
\]

So it suffices to show that

(a) \( \text{defs}_1 \text{let } x_1 = e_1 \text{ in } e_2 = (\text{defs}_1[e_1], x_1 = \text{hole}[e_1], \text{defs}_2[e_2]) \)

(b) \( \text{hole} \text{let } x_1 = e_1 \text{ in } e_2 = \text{hole}[e_2] \)

both of which are straightforward by definition.

\( \square \)

**Theorem 3.10 (Type Preservation).** If \( \Gamma \vdash e : A \) then \( [\Gamma] \vdash [e] : [A] \)

**Proof.** By Lemma 3.9, it suffices to show that \( [\Gamma] \vdash [\cdot] : [\cdot : [A]] \Rightarrow [A], \) which is trivial.

\( \square \)

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, \textit{i.e.}, we would
\( v \approx V \)

\[
\begin{align*}
\text{true} & \approx \text{true} \\
\text{false} & \approx \text{false}
\end{align*}
\]

\( \Gamma \vdash \gamma \)

\[
\begin{align*}
\cdot \vdash \emptyset & \quad \Gamma, x \equiv e \vdash \gamma[x \mapsto \gamma(e)] \\
\cdot \vdash e : A & \quad \Gamma \vdash \gamma \\
\Gamma, x : A \vdash \gamma[x \mapsto e]
\end{align*}
\]

Fig. 13. Separate Compilation Definitions

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 13.

We define linking as substitution with well-typed closed terms, and define a closing substitution \( \gamma \) with respect to the environment \( \Gamma \) (also in Figure 13). Linking is defined by closing a term \( e \) such that \( \Gamma \vdash e : A \) with a substitution \( \Gamma \vdash \gamma \), written \( \gamma(e) \). Any \( \gamma \) 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 \( \gamma[x \mapsto \gamma(e)] \), 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 3.11 (Correctness of Separate Compilation). If \( \Gamma \vdash e : A \) (and \( A \) ground) and \( \Gamma \vdash \gamma \) then \( \text{eval}([\gamma] ([e])) \approx \text{eval}(\gamma(e)) \).

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.

\[
\begin{array}{c}
\text{eval}(\gamma(e)) \xrightarrow{\equiv} [\gamma(e)] \\
\text{eval}([\gamma([e])]) \xrightarrow{\equiv} [\gamma([e])]
\end{array}
\]

\( \square \)
REFERENCES


