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) [Luo 1990] extended with dependent elimination of booleans and with definitions [Severi and Poll 1994]. We typeset ECC in a non-bold, blue, sans-serif font. We present the syntax of ECC in Figure 1. ECC extends the Calculus of Constructions (CC) [Coquand and Huet 1988] with Σ types (strong dependent pairs) and an infinite predicative hierarchy of universes. There is no explicit phase distinction, *i.e.*, there is no syntactic distinction between *terms*, which represent run-time expressions, and *types*, which classify terms. However, we usually use the meta-variable e to evoke a term, and the meta-variables A and B to evoke a type. The language includes one impredicative universe, Prop, and an infinite hierarchy of predicative universes Type_i. The syntax of expressions e includes names x, universes U, dependent function types $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, functions $\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}$, application $\mathsf{e}_1\ \mathsf{e}_2$, dependent pair types $\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, dependent pairs $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ as $\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, first $\mathsf{fst}\ \mathsf{e}$ and second $\mathsf{snd}\ \mathsf{e}$ projections of dependent pairs, dependent let $\mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}'$, the boolean type bool, the boolean value true and false, and dependent if $\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$. For brevity, we omit the type annotation on dependent pairs, as in $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$. Note that let-bound definitions do not include type annotations; this is not standard, but type checking is still decidable [Severi and Poll 1994], and it simplifies our ANF translation[1].

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

In Figure 2, we give the reductions $\Gamma \vdash \mathsf{e} \rhd \mathsf{e}'$ for ECC, which are entirely standard. As usual, we extend reduction to conversion by defining $\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'$ to be the reflexive, transitive, compatible closure of reduction $\rhd$. 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 $\mathsf{eval}(\mathsf{e})$ as the evaluation function for whole-programs using conversion, which we use in our compiler correctness proof.

In Figure 4 we define *definitional equivalence* (or just *equivalence*) $\Gamma \vdash \mathsf{e} \equiv \mathsf{e}'$ as conversion up to η-equivalence. We usually we the notation $\mathsf{e}_1 \equiv \mathsf{e}_2$ for equivalence, eliding the environment when it is obvious or unnecessary. We also define *cumulativity* (subtyping) $\Gamma \vdash \mathsf{A} \preceq \mathsf{B}$, 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 $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ have types (called universes), and universes also have types which are higher universes. In [Ax-Prop], the type of Prop is Type_0, and in [Ax-Type], the type of each universe Type_i is the next higher universe Type_{i+1}. Note that we have impredicative function types in Prop, given by [Prod-Prop]. For this work, we ignore the Set vs Prop distinction used in some type theories, such as Coq's, although adding it causes no difficulty. Note that the rules for application, [App], second projection, [Snd], let, [Let], and if [If] substitute sub-expressions into the type system. These are the key typing rules that introduce difficulty in type-preserving compilation for dependent types.

[*]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.

$$
\begin{array}{llll}
\textit{Universes} & \mathsf{U} & ::= & \mathsf{Prop} \mid \mathsf{Type}_i \\
\textit{Expressions} & \mathsf{e}, \mathsf{A}, \mathsf{B} & ::= & \mathsf{x} \mid \mathsf{U} \mid \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \mid \lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e} \mid \mathsf{e}\;\mathsf{e} \mid \Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle\;\mathsf{as}\;\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \\
 & & \mid & \mathsf{fst}\;\mathsf{e} \mid \mathsf{snd}\;\mathsf{e} \mid \mathsf{let}\;\mathsf{x} = \mathsf{e}\;\mathsf{in}\;\mathsf{e} \mid \mathsf{bool} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\;\mathsf{e}\;\mathsf{then}\;\mathsf{e}_1\;\mathsf{else}\;\mathsf{e}_2 \\
\textit{Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x} : \mathsf{A} \mid \Gamma, \mathsf{x} = \mathsf{e}
\end{array}
$$

Fig. 1. ECC Syntax

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

$$
\begin{array}{rcll}
\mathsf{x} & \rhd_{\delta} & \mathsf{e} & \text{where } \mathsf{x} = \mathsf{e} \in \Gamma \\
(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1)\;\mathsf{e}_2 & \rhd_{\beta} & \mathsf{e}_1[\mathsf{x} := \mathsf{e}_2] & \\
\mathsf{fst}\;\langle \mathsf{e}_1, \mathsf{e}_2 \rangle & \rhd_{\pi_1} & \mathsf{e}_1 & \\
\mathsf{snd}\;\langle \mathsf{e}_1, \mathsf{e}_2 \rangle & \rhd_{\pi_2} & \mathsf{e}_2 & \\
\mathsf{let}\;\mathsf{x} = \mathsf{e}\;\mathsf{in}\;\mathsf{e}' & \rhd_{\zeta} & \mathsf{e}'[\mathsf{x} := \mathsf{e}] & \\
\mathsf{if}\;\mathsf{true}\;\mathsf{then}\;\mathsf{e}_1\;\mathsf{else}\;\mathsf{e}_2 & \rhd_{\iota_1} & \mathsf{e}_1 & \\
\mathsf{if}\;\mathsf{false}\;\mathsf{then}\;\mathsf{e}_1\;\mathsf{else}\;\mathsf{e}_2 & \rhd_{\iota_2} & \mathsf{e}_2 &
\end{array}
$$

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

$$
\ldots \qquad \frac{\Gamma, \mathsf{x} = \mathsf{e} \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_2}{\Gamma \vdash \mathsf{let}\;\mathsf{x} = \mathsf{e}\;\mathsf{in}\;\mathsf{e}_1 \rhd^* \mathsf{let}\;\mathsf{x} = \mathsf{e}\;\mathsf{in}\;\mathsf{e}_2}\;[\textsc{Red-Cong-Let}] \qquad \frac{}{\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}}\;[\textsc{Red-Refl}]
$$

$$
\frac{\Gamma \vdash \mathsf{e} \rhd \mathsf{e}_1 \qquad \Gamma \vdash \mathsf{e}_1 \rhd^* \mathsf{e}'}{\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'}\;[\textsc{Red-Trans}]
$$

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

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

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

$$\frac{\Gamma \vdash A \rhd^* A' \qquad \Gamma, x : A \vdash e \rhd^* e'}{\Gamma \vdash \lambda\, x : A.\, e \rhd^* \lambda\, x : A'.\, e'}\ [\textsc{Red-Cong-Lam1}] \qquad \frac{\Gamma \vdash A \rhd^* A' \qquad \Gamma, x : A \vdash B \rhd^* B'}{\Gamma \vdash \Pi\, x : A.\, B \rhd^* \Pi\, x : A'.\, B'}\ [\textsc{Red-Cong-Pi}]$$

$$\frac{\Gamma \vdash A \rhd^* A' \qquad \Gamma, x : A \vdash B \rhd^* B'}{\Gamma \vdash \Sigma\, x : A.\, B \rhd^* \Sigma\, x : A'.\, B'}\ [\textsc{Red-Cong-Sig}]$$

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2' \qquad \Gamma \vdash A \rhd^* A'}{\Gamma \vdash \langle e_1, e_2 \rangle \text{ as } A \rhd^* \langle e_1', e_2' \rangle \text{ as } A'}\ [\textsc{Red-Cong-Pair}] \qquad \frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2'}{\Gamma \vdash e_1\ e_2 \rhd^* e_1'\ e_2'}\ [\textsc{Red-Cong-App}]$$

$$\frac{\Gamma \vdash V \rhd^* V'}{\Gamma \vdash \text{fst } V \rhd^* \text{fst } V'}\ [\textsc{Red-Cong-Fst}] \qquad \frac{\Gamma \vdash V \rhd^* V'}{\Gamma \vdash \text{snd } V \rhd^* \text{snd } V'}\ [\textsc{Red-Cong-Snd}]$$

$$\frac{\Gamma, x = N \vdash M \rhd^* M'}{\Gamma \vdash \text{let } x = N \text{ in } M \rhd^* \text{let } x = N \text{ in } M'}\ [\textsc{Red-Cong-Let}]$$

$$\frac{\Gamma \vdash e \rhd^* e' \qquad \Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \rhd^* \text{if } e' \text{ then } e_1' \text{ else } e_2'}\ [\textsc{Red-Cong-If}]$$

Fig. 3. ECC Congruence Conversion Rules

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

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

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

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

$$\frac{\Gamma \vdash A \equiv B}{\Gamma \vdash A \preceq B}\ [\preceq\text{-}\equiv] \qquad \frac{\Gamma \vdash A \preceq A' \qquad \Gamma \vdash A' \preceq B}{\Gamma \vdash A \preceq B}\ [\preceq\text{-}\textsc{Trans}] \qquad \frac{}{\Gamma \vdash \text{Prop} \preceq \text{Type}_0}\ [\preceq\text{-}\textsc{Prop}]$$

$$\frac{}{\Gamma \vdash \text{Type}_i \preceq \text{Type}_{i+1}}\ [\preceq\text{-}\textsc{Cum}] \qquad \frac{\Gamma \vdash A_1 \equiv A_2 \qquad \Gamma, x_1 : A_2 \vdash B_1 \preceq B_2[x_2 := x_1]}{\Gamma \vdash \Pi\, x_1 : A_1.\, B_1 \preceq \Pi\, x_2 : A_2.\, B_2}\ [\preceq\text{-}\textsc{Pi}]$$

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

Fig. 4. ECC Equivalence and Subtyping

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

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

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

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

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

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

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

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

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

Fig. 5. ECC Typing

$\boxed{\vdash \Gamma}$

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

Fig. 6. ECC Well-Formed Environments

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

Fig. 7. ECC^A Syntax

. . .

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

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

Fig. 8. ECC^A Typing (excerpts)

2 TARGET: ECC WITH ANF SUPPORT

Our target language, ECC^A, is a variant of ECC with a modified typing rule for dependent **if** that introduces equalities between terms (akin to the identity type), and an elimination form for assumed equalities. These extensions are similar to the extensions used by Cong and Asai [2018] to support CPS translation with dependent pattern matching. While ECC^A supports ANF syntax, the full language is not ANF restricted; it has the same syntax as ECC, and uses the usual definitional equivalence and conversion relation for type checking. We do not restrict the full language because maintaining ANF while type checking adds needless complexity; instead, we show that our compiler generates only ANF restricted terms in ECC^A, and define a separate ANF-preserving machine-like semantics for evaluating programs in ANF.

We can imagine the compilation process as either: (1) generating ANF syntax in ECC^A from ECC, or (2) as first embedding ECC in ECC^A and then rewriting ECC^A terms into ANF. In Section 3 we present the compiler as process (1), a compiler from ECC to ANF ECC^A. In this section we develop most of the supporting meta-theory necessary for ANF as intra-language equivalences and process (2) may be a more helpful intuition. We typeset ECC^A in a **bold, red, serif font**; in later sections, we reserve this font exclusively for the ANF restricted ECC^A.

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

We give the ANF syntax for ECC^A in Figure 7. We impose a syntactic distinction between *values* $\mathbf{V}$ which do not reduce, *computations* $\mathbf{N}$ which eliminate values and can be composed using *continuations* $\mathbf{K}$, and *configurations* $\mathbf{M}$ which represent the machine configurations executed by the ANF machine semantics. A continuation $\mathbf{K}$ is a program with a hole, and is composed $\mathbf{K}[\mathbf{N}]$ with a computation $\mathbf{N}$ to form a configuration $\mathbf{M}$. For example, $(\mathbf{let}\ \mathbf{x}=[\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x})[\mathbf{N}] = (\mathbf{let}\ \mathbf{x}=\mathbf{N}\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x})$. Since continuations are not first-class objects, we cannot express control effects—continuations are syntactically guaranteed to be used linearly. Note that despite the *syntactic* distinctions, we still do not enforce a *phase* distinction—configurations (programs) can appear in types.

We give the new typing rules in Figure 8. The key change in ECC^A is in the typing rule for dependent if. The typing rule for $\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2$ introduces an unnamed equality in each branch. This records the information that in $\mathbf{e}_1$ the target $\mathbf{e}$ being eliminated will be equal to $\mathbf{true}$ ($\mathbf{e} = \mathbf{true}$), and in $\mathbf{e}_2$ we know statically that $\mathbf{e} = \mathbf{false}$. These

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

$$\cdots \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \mathsf{subst}_{e_1=e_2}\, e \equiv e'}\ [\equiv\text{-Subst}_1] \qquad \frac{\mathsf{true} = \mathsf{false} \in \Gamma}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Absurd}_1] \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash e' \equiv \mathsf{subst}_{e_1=e_2}\, e}\ [\equiv\text{-Subst}_2]$$

$$\frac{\mathsf{false} = \mathsf{true} \in \Gamma}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Absurd}_2]$$

Fig. 9. ECC^A Equivalence (excerpts)

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

$$\begin{aligned}
K\langle\langle N\rangle\rangle &\overset{\text{def}}{=} K[N] \\
K\langle\langle \mathsf{let}\ x = N'\ \mathsf{in}\ M\rangle\rangle &\overset{\text{def}}{=} \mathsf{let}\ x = N'\ \mathsf{in}\ K\langle\langle M\rangle\rangle \\
K\langle\langle \mathsf{if}\ V\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2\rangle\rangle &\overset{\text{def}}{=} \mathsf{if}\ V\ \mathsf{then}\ (K[\mathsf{subst}_{V=\mathsf{true}}\ x][M_1//x])\ \mathsf{else}\ (K[\mathsf{subst}_{V=\mathsf{false}}\ x][M_2//x])
\end{aligned}$$

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

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

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

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

Fig. 10. Composition of Configurations

record a machine step: the machine will have reduced e to **true** before jumping to the first branch, and reduced e to **false** before jumping to the second branch. This is necessary to support the ANF transformation of dependent if.

The typing rule for **subst** is essentially a standard eliminator of the identity type. The typing rule for an expression $\mathsf{subst}_{e_1=e_2}\ e$ replaces e_1 by e_2 in the type of e, if we have assumed $e_1 = e_2$. To ensure normalization, $\mathsf{subst}_{e_1=e_2}\ e$ can only reduce when e_1 and e_2 are syntactically identical, analogous to requiring the canonical proof **refl** of the identity type. We use unnamed equality assumptions instead of the identity type to simplify the syntax of dependent if.

We require several additional equivalence rules; the two key rules are shown in Figure 9. These rules are necessary for proving ANF is type preserving in the presence of dependent if. The rule $[\equiv\text{-Subst}_1]$ (and a symmetric rule $[\equiv\text{-Subst}_2]$) allows reasoning about equivalence of **subst** without reducing it, maintaining normalization in the presence of possibly inconsistent assumed equalities. The second rule is required in exactly two places in the type-preservation proof, and allows us to conclude any two terms are equivalent if we assume **true** = **false** (or, symmetrically, **false** = **true**).[2]

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 $K[M[x := V]]$. Substituting the term $M[x := V]$, a *configuration*, into the continuation K could result in the non-ANF term $\mathsf{let}\ x = M\ \mathsf{in}\ M'$. In ANF, configurations cannot be nested.

To ensure reduction preserves ANF, we define composition of a continuation K and a configuration M, Figure 10, typically called *renormalization* in the literature [Kennedy 2007; Sabry and Wadler 1997]. When composing a continuation with a configuration, $K\langle\langle M\rangle\rangle$, we essentially unnest all continuations so they remain left associated. [3] When composing a continuation with an **if** statement, notice that the continuation is duplicated in the branches. This is the

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

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

$\boxed{M \mapsto M'}$

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

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

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

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

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

Fig. 11. ECC^A Evaluation

usual, naïve, presentation of ANF; we show later that the join-point optimization, which avoids this duplication, is admissible in ECC^A. Note that these definitions are simplified by our uniqueness-of-names assumption.

Digression on composition in ANF. In the literature, the composition operation $K\langle\!\langle M\rangle\!\rangle$ is usually introduced as *renormalization*, as if the only intuition for why it exists is "well, it happens that ANF is not preserved under β-reduction". It is not mere coincidence; the intuition for this operation is composition, and having a syntax for composing terms is not only useful for stating β-reduction, but useful for all reasoning about ANF! This should not come as a surprise—compositional reasoning is useful. The only surprise is that the composition operation is not the usual one used in programming language semantics, *i.e.*, substitution. In ANF, as in monadic normal form, substitution can be used to compose any expression with a *value*, since names are values and values can always be replaced by values. But substitution cannot just replace a name, which is a *value*, with a *computation* or *configuration*. That wouldn't be well-typed. So how do we compose computations with configurations? We can use **let**, as in $\text{let}\ y = N\ \text{in}\ M$, 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 $\text{let}\ y = [\cdot]\ \text{in}\ M'$, 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: $K\langle\!\langle M\rangle\!\rangle$. This operation is more generally known as *hereditary substitution* [Watkins et al. 2003], a form of substitution that maintains canonical forms. So we can think of it as a form of substitution, or, simply, as composition.

In Figure 11, we present the call-by-value (CBV) evaluation semantics for ANF ECC^A terms. It is essentially standard, but recall that β-reduction produces a configuration M which must be composed with the existing continuation K. This semantics is for the run-time evaluation of configurations; during type checking, we continue to use the type system and conversion relation defined in Section 1.

2.1 Dependent Continuation Typing

The ANF translation manipulates continuations K as independent entities. To reason about them, and thus to reason about the translation, we introduce continuation typing, defined in Figure 12. The type $(M : A) \Rightarrow B$ of a continuation expresses that this continuation expects to be composed with a term equal to the configuration M of type A and returns a result of type B when completed. Normally, M is equivalent to some computation N, but it must be generalized to a configuration M to support dependent **if** expressions. This type formally expresses the idea that M is depended upon

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

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

Fig. 12. ECC^A Continuation Typing

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

$$\begin{array}{rcl}
[\![\mathbf{subst}_{\,e_1 = e_2}\ e]\!]_M & \overset{\text{def}}{=} & \text{subst}\ p\ [\![e]\!]_M \\
& & \text{where } p : [\![e_1]\!]_M = [\![e_2]\!]_M \in [\![\Gamma]\!]_M \\
[\![\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]_M & \overset{\text{def}}{=} & (\text{if } [\![e]\!]_M \text{ then } (\lambda\, p : \text{true} = [\![e]\!]_M.\ \text{subst (if-eta1 } [\![e_1]\!]_M\ [\![e_2]\!]_M\ p)\ [\![e_1]\!]_M) \\
& & \qquad\qquad\quad \text{else } (\lambda\, p : \text{false} = [\![e]\!]_M.\ \text{subst (if-eta2 } [\![e_1]\!]_M\ [\![e_2]\!]_M\ p)\ [\![e_2]\!]_M)) \\
& & \text{refl } [\![e]\!]_M
\end{array}$$

$$\boxed{\text{Auxiliary CIC definitions}}$$

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

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

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

Fig. 13. ECC^A Model in CIC (excerpts)

(in the sense introduced in ??) in the rest of the computation. For the empty continuation $[\cdot]$, M is arbitrary since an empty continuation has no "rest of the program" that could depend on anything.

Intuitively, what we want from continuation typing is a compositionality property—that we can reason about the types of configurations $K[N]$ (generally, for configurations $K\langle\!\langle M\rangle\!\rangle$) by composing the typing derivations for K and N. To get this property, a continuation type must express not merely the *type* of its hole A, but *which term* N will be bound in the hole. We see this formally from the typing rule [Let] (the same for ECC^A as for ECC in Section 1), since showing that $\mathbf{let}\ y = N\ \mathbf{in}\ M$ is well-typed requires showing that $y = N \vdash M$, 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 2.8 (Cut).

Note that the result of a continuation type cannot depend on the term that will be plugged in for the hole, *i.e.*, for a continuation $K : (N : A) \Rightarrow B$, B does not depend on N. To see this, first note that the initial continuation must be empty and thus *cannot* have a result type that depends on its hole. The ANF translation will take this initial empty continuation and compose it with intermediate continuations K'. Since composing any continuation $K : (N : A) \Rightarrow B$ 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 $\bot$.

2.2 Meta-Theory

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

The essence of the model is given in Figure 13. There are only two interesting rules. The form $\mathbf{subst}_{\mathbf{e}_1=\mathbf{e}_2}\ \mathbf{e}$ is simply translated into a call to the function subst, implemented in CIC, applied to some variable p of the identity type that is in scope in the model. Each if expression $\mathbf{if\ e\ then\ e_1\ else\ e_2}$ is implemented using the convoy pattern, transforming each if expression into a new if expression that returns a function expecting a proof that $[\![\mathbf{e}]\!]_M$ is equal to true in the first branch and false in the second branch. The if expression is then immediately applied to refl. The model relies on auxiliary definitions in CIC, including subst, if-eta1 and if-eta2, whose types are given as inference rules in Figure 13. Note that the model for ECC^A's $\mathbf{if}$ is not valid ANF, so it does not suffice to merely *use* the convoy pattern if we want to take advantage of ANF for compilation.

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

We use the usual definition of $\mathbf{False}$ as $\mathbf{\Pi\, x : Prop\,.\, x}$, and the same in CIC. It is trivial that the definition is preserved.

Lemma 2.1 (Model Preserves Falseness). $[\![\mathbf{False}]\!]_M \equiv \mathit{False}$

The essence of showing both that equivalence is preserved and that typing is preserved is in showing that the auxiliary definitions in Figure 13 exist and are well typed. We define these formally in the Coq model. Note, however, that Lemma 2.2 is stated in terms of definitional equivalence, while the Coq implementation uses propositional equivalence. This means interpreting our Coq implementation as a model requires equivalence reflection.

Lemma 2.2 (Model Preserves Equivalence). *If* $\mathbf{e_1} \equiv \mathbf{e_2}$ *then* $[\![\mathbf{e_1}]\!]_M \equiv [\![\mathbf{e_2}]\!]_M$.

Lemma 2.3 (Model Preserves Typing). *If* $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ *then* $[\![\mathbf{\Gamma}]\!]_M \vdash [\![\mathbf{e}]\!]_M : [\![\mathbf{A}]\!]_M$.

Theorem 2.4 (Consistency). *There is no* $\mathbf{e}$ *such that* $\cdot \vdash \mathbf{e} : \mathbf{False}$

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

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

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

Lemma 2.5 is the formal statement of naturality in ANF: composing a term $\mathbf{M}$ with its continuation $\mathbf{K}$ in ANF is equivalent to running $\mathbf{M}$ to a value and substituting the result into the continuation $\mathbf{K}$. Formally, this states that composing continuations in ANF is sound with respect to standard substitution.

Lemma 2.5 (Naturality). $\mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle \equiv \mathcal{ST}(\mathbf{K}[\mathbf{M}])$

Proof. By induction on the structure of $\mathbf{M}$

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

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

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

□

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

Lemma 2.6 (Small-step soundness). *If* $\mathrm{M} \mapsto \mathrm{M}'$ *then* $\mathrm{M} \equiv \mathrm{M}'$

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

Case: $\mathrm{K}[(\lambda\,\mathrm{x} : \mathrm{A}.\,\mathrm{M}_1)\;\mathrm{V}] \mapsto_{\beta} \mathrm{K}\langle\langle \mathrm{M}_1[\mathrm{x} := \mathrm{V}]\rangle\rangle$
Must show that $\mathrm{K}[(\lambda\,\mathrm{x} : \mathrm{A}.\,\mathrm{M}_1)\;\mathrm{V}] \equiv \mathrm{K}\langle\langle \mathrm{M}_1[\mathrm{x} := \mathrm{V}]\rangle\rangle$

$$\begin{aligned}
&\mathrm{K}[(\lambda\,\mathrm{x} : \mathrm{A}.\,\mathrm{M}_1)\;\mathrm{V}] \\
\triangleright^{*}\; & \mathcal{ST}(\mathrm{K}[\mathrm{M}_1[\mathrm{x} := \mathrm{V}]]) && \text{by } \beta \text{ and congruence} \quad (4)\\
\equiv\; & \mathrm{K}\langle\langle \mathrm{M}_1[\mathrm{x} := \mathrm{V}]\rangle\rangle && \text{by Lemma 2.5} \quad (5)
\end{aligned}$$

Case: $\mathrm{K}[\mathrm{fst}\,\langle \mathrm{V}_1, \mathrm{V}_2\rangle] \mapsto_{\pi_1} \mathrm{K}[\mathrm{V}_1]$
Must show that $\mathrm{K}[\mathrm{fst}\,\langle \mathrm{V}_1, \mathrm{V}_2\rangle] \equiv \mathrm{K}[\mathrm{V}_1]$, which follows by $\triangleright_{\pi_1}$ and congruence. □

Theorem 2.7 (Evaluation soundness). $\vdash \mathrm{eval}(\mathrm{M}) \equiv \mathrm{M}$

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

Case: $n = 0$ By [Red-Refl] and [≡].
Case: $n = i + 1$ Follows by Lemma 2.6 and the induction hypothesis. □

2.2.3 Admissibility of Continuation Typing. To prove that continuation typing is not an extension to the type system—*i.e.*, is admissible—we prove Lemma 2.8 and Lemma 2.12, that plugging a well-typed computation or configuration into a well-typed continuation results in a well-typed term of the expected type.

We first show Lemma 2.8 (Cut), which is simple. This lemma corresponds to the [Cut] rule (*e.g.*, found in sequent calculi), and tells us that our continuation typing allows for compositional reasoning about configurations $\mathrm{K}[\mathrm{N}]$ whose result types do not depend on N. We generalize this lemma to configurations $\mathrm{K}\langle\langle \mathrm{N}\rangle\rangle$ shortly, but require more meta-theory to do so. The proof for this lemma is simple by definition.

Lemma 2.8 (Cut). *If* $\Gamma \vdash \mathrm{K} : (\mathrm{N} : \mathrm{A}) \Rightarrow \mathrm{B}$ *and* $\Gamma \vdash \mathrm{N} : \mathrm{A}$ *then* $\Gamma \vdash \mathrm{K}[\mathrm{N}] : \mathrm{B}$.

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

Case: $\Gamma \vdash [\cdot] : (\mathrm{N} : \mathrm{A}) \Rightarrow \mathrm{A}$, trivial
Case: $\Gamma \vdash \mathrm{let}\,\mathrm{y} = [\cdot]\,\mathrm{in}\,\mathrm{M} : (\mathrm{N} : \mathrm{A}) \Rightarrow \mathrm{B}$
We must show that $\Gamma \vdash \mathrm{let}\,\mathrm{y} = \mathrm{N}\,\mathrm{in}\,\mathrm{M} : \mathrm{B}$, which follows directly from [Let] since, by the continuation typing derivation, we have that $\Gamma, \mathrm{y} = \mathrm{N} \vdash \mathrm{M} : \mathrm{B}$ and $\mathrm{y} \notin \mathrm{fv}(\mathrm{B})$. □

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

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

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

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

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

Fig. 14. Continuation Exports

Lemma 2.9 (Cut Modulo Equivalence). *If* $\Gamma \vdash K : (N : A) \Rightarrow B$, $\Gamma \vdash N : A$, $\Gamma \vdash N' : A$, *and* $\Gamma \vdash N \equiv N'$, *then* $\Gamma \vdash K[N'] : B$.

Proof. By cases on the structure of K.

Case: $K = [\cdot]$. Trivial.

Case: $K = \textbf{let}\ x = [\cdot]\ \textbf{in}\ M'$

It suffices to show that: If $\Gamma, x = N \vdash M' : B$ then $\Gamma, x = N' \vdash M' : B$.

Note that anywhere in the derivation $\Gamma, x = N \vdash M' : B$ that $x = N$ is used, it must be used essentially as: $A \equiv_\zeta A[x := N]$. We can replace any such use by $A \equiv_\zeta A[x := N'] \equiv A[x := N]$ to construct a derivation for $\Gamma, x = N' \vdash M' : B$

□

To reason inductively about ANF terms, we need to separate a configuration M into its exported definitions $\texttt{defs}(M)$ and its underlying computation $\texttt{hole}(M)$, which we define formally in Figure 14. The exported definitions represent all the machine steps (**let**-bound computations) that will happen "before" (in CBV, anyway) executing the body of M, while the $\texttt{hole}(M)$ is the body, the inner-most computation whose result will be bound when M is composed with another continuation. We define $\texttt{defs}(M)$ to be the sequence of definitions bound in the ANF term M. These are the definitions that will be in scope for a continuation K when composed with M, *i.e.*, in scope for K in $K\langle\langle M\rangle\rangle$. Note that $\texttt{hole}(M)$ will only be well typed in the environment for M extended with the definitions $\texttt{defs}(M)$.

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

Lemma 2.10. *If* $\Gamma \vdash M : A$ *then* $\Gamma, \texttt{defs}(M) \vdash M \rhd^* \texttt{hole}(M)$

Proof. By induction on the syntax of M.

Case: N

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

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

Must show that $\Gamma, x = N, \texttt{defs}(M') \vdash \textbf{let}\ x = N\ \textbf{in}\ M' \rhd^* \texttt{hole}(M')$.

By [Red-Cong-Let], it suffices to show $\Gamma, x = N, \texttt{defs}(M'), x = N \vdash M' \rhd^* \texttt{hole}(M')$; recall that [Red-Cong-Let] introduces a redundant definition $x = N$.

Since names are unique, the second $x = N$ is redundant, and $\Gamma, x = N, \texttt{defs}(M') \vdash M' \rhd^* \texttt{hole}(M')$ follows by the induction hypothesis.

Case: $\textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2$

Must show that $\Gamma \vdash \textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \rhd^* \textbf{if}\ V\ \textbf{then}\ \texttt{hole}(M_1)\ \textbf{else}\ \texttt{hole}(M_2)$.

which follows by the congruence rule for if and the induction hypothesis. □

Corollary 2.11. *If* $\Gamma \vdash M : A$ *then* $\Gamma, \texttt{defs}(M) \vdash M \equiv \texttt{hole}(M)$.

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

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

This suggests we could define continuation typing as follows.

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

That is, instead of adding separate rules [K-Empty] and [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 2.8 (Cut) is definitional rather than admissible. This rule is somewhat surprising; it appears very much like the definition of [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 [K-Type] is related to the definitions we wish to focus on. It is exactly the premises of [K-Bind] that we need to recover type-preservation for ANF, so we choose the presentation with [K-Bind].

However, the rule [K-Type] is more general in the sense that the continuation typing does not need and changes as the definition of continuations change.

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

Lemma 2.12 (Heterogeneous Cut). *If* $\Gamma \vdash M : B'$ *and* $\Gamma, \mathtt{defs}(M) \vdash \mathtt{hole}(M) : B$ *and* $\Gamma, \mathtt{defs}(M) \vdash K : (\mathtt{hole}(M) : B) \Rightarrow C$ *then* $\Gamma \vdash K\langle\!\langle M\rangle\!\rangle : C$.

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

Case: $\Gamma \vdash N : M$, by Lemma 2.8.

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

By the premise, we know:

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

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

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

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

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

Case: $\Gamma \vdash \mathbf{if}\, V \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_2 : B''[x := V]$ By the premise, we know:

(a) $\Gamma, V = \mathbf{true} \vdash M_1 : B''[x := \mathbf{true}]$ by inversion

(b) $\Gamma, V = \mathbf{false} \vdash M_2 : B''[x := \mathbf{false}]$ by inversion

(c) $\Gamma \vdash \mathbf{if}\, V \,\mathbf{then}\, \mathtt{hole}(M_1) \,\mathbf{else}\, \mathtt{hole}(M_2) : B[x := V]$ by definition

(d) $\Gamma \vdash K : (\mathbf{if}\, V \,\mathbf{then}\, \mathtt{hole}(M_1) \,\mathbf{else}\, \mathtt{hole}(M_2) : B[x := V]) \Rightarrow C$ by definition

By [If], it suffices to show

(a) $\Gamma, V = \mathbf{true} \vdash K[\mathbf{subst}_{V=\mathbf{true}}\, x][M_1 // x] : C$

By definition, it suffices to show $\Gamma, V = \mathbf{true} \vdash (\mathbf{let}\, x = [\cdot] \,\mathbf{in}\, K[\mathbf{subst}_{V=\mathbf{true}}\, x])\langle\!\langle M_1\rangle\!\rangle : C$.

By the induction hypothesis, it suffices to show that: $\Gamma, V = \mathbf{true}, \mathtt{defs}(M_1) \vdash (\mathbf{let}\, x = [\cdot] \,\mathbf{in}\, K[\mathbf{subst}_{V=\mathbf{true}}\, x]) : (\mathtt{hole}(M_1) : B''[x := \mathbf{true}]) \Rightarrow C$.

By continuation typing, it suffices to show

$\Gamma, V = \mathbf{true}, \mathtt{defs}(M_1), x = \mathtt{hole}(M_1) \vdash (K[\mathbf{subst}_{V=\mathbf{true}}\, x]) : C$, which follows by Lemma 2.8 since

$\Gamma, V = \mathbf{true}, \mathtt{defs}(M_1), x = \mathtt{hole}(M_1) \vdash \mathbf{subst}_{V=\mathbf{true}}\, x \equiv \mathbf{if}\, V \,\mathbf{then}\, \mathtt{hole}(M_1) \,\mathbf{else}\, \mathtt{hole}(M_2)$.

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

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

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

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

Fig. 15. Naïve ANF Translation

3 ANF TRANSLATION

The ANF translation is presented in Figure 15. The naïve translation is defined inductively over syntax. The translation is indexed by a current continuation, which is used when translating a value and is is composed together "inside-out" the same continuation composition is defined in Section 2. The translation is essentially standard. When translating a value such as x, $\lambda\, x : A.\, e$, and Type_i, we essentially plug the value into the current continuation, recursively translating the sub-expressions of the value if applicable. For non-values such as application, we make sequencing explicit by recursively translating each sub-expression with a continuation that binds the result which will perform the computation.

Note that if the translation must produce type annotations then defining the translation and typing preservation proof are somewhat more complicated. For instance, if we generate join points directly, or require the **let**-bindings in the target language to have type annotations for bound expressions, then we would need to modify the translation to produce those annotations. This requires defining the translation over typing derivations, so the compiler has access to the type of the expression and not only its syntax.

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

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 from the meta-theory about the machine semantics proved in Section 2, and from equivalence preservation.

Recall from Section 2, we shift from the **target language font** to the source language font whenever we shift out of ANF, such as when we perform standard substitution or conversion. When the shift in font is not apparent, we use the language boundary term $\mathcal{ST}()$.

Before we proceed, we state a property about the syntactic form produced by the translation, in particular, that the ANF translation does produce syntax in ANF (Theorem 3.1). The proof is straightforward so we elide it.

Theorem 3.1 (ANF). *For all* e *and* K, $[\![\mathrm{e}]\!]\ \mathrm{K} = \mathrm{M}$ *for some* M.

As discussed in Section 2, composition in ANF is somewhat non-standard. Normally, we compose via substitution, so the compositionality property we want is $[\![\mathrm{e}[\mathrm{x} := \mathrm{e}']]\!] \equiv [\![\mathrm{e}]\!][\mathrm{x} := [\![\mathrm{e}']\!]]$, 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). $\mathrm{K}'\langle\!\langle\ [\![\mathrm{e}]\!]\ \mathrm{K}\rangle\!\rangle = [\![\mathrm{e}]\!]\ \mathrm{K}'\langle\!\langle \mathrm{K}\rangle\!\rangle$

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: $\mathrm{e} = \mathrm{x}$ Must show $\mathrm{K}'\langle\!\langle \mathrm{K}[\mathrm{x}]\rangle\!\rangle = \mathrm{K}'\langle\!\langle \mathrm{K}[\mathrm{x}]\rangle\!\rangle$, which is trivial.

Case: $\mathrm{e} = \Pi\,\mathrm{x} : \mathrm{A}.\,\mathrm{B}$ Must show that $\mathrm{K}'\langle\!\langle \mathrm{K}[\Pi\,\mathrm{x} : [\![\mathrm{A}]\!].\,[\![\mathrm{B}]\!]]\rangle\!\rangle = \mathrm{K}'\langle\!\langle \mathrm{K}[\Pi\,\mathrm{x} : [\![\mathrm{A}]\!].\,[\![\mathrm{B}]\!]]\rangle\!\rangle$, which is trivial. Note that we need not appeal to induction, since the recursive translation does not use the current continuation for values.

Case: $\mathrm{e} = \mathrm{e}_1\ \mathrm{e}_2$ Must show that

$$\begin{aligned}&\mathrm{K}'\langle\!\langle([\![\mathrm{e}_1]\!]\ (\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathrm{e}_2]\!]\ \mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}[\mathrm{x}_1\ \mathrm{x}_2])))\rangle\!\rangle\\ &= ([\![\mathrm{e}_1]\!]\ (\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathrm{e}_2]\!]\ \mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}'\langle\!\langle \mathrm{K}\rangle\!\rangle[\mathrm{x}_1\ \mathrm{x}_2])))\end{aligned}$$

The proof follows essentially from the definition of continuation composition.

$$\begin{aligned}
& && \mathrm{K}'\langle\!\langle([\![\mathrm{e}_1]\!]\ (\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathrm{e}_2]\!]\ \mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}[\mathrm{x}_1\ \mathrm{x}_2])))\rangle\!\rangle \\
&= && ([\![\mathrm{e}_1]\!]\ \mathrm{K}'\langle\!\langle(\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathrm{e}_2]\!]\ \mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}[\mathrm{x}_1\ \mathrm{x}_2]))\rangle\!\rangle) && (6)\\
& && \text{by the induction hypothesis applied to } \mathrm{e}_1 \\
&= && ([\![\mathrm{e}_1]\!]\ (\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ \mathrm{K}'\langle\!\langle([\![\mathrm{e}_2]\!]\ \mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}[\mathrm{x}_1\ \mathrm{x}_2])\rangle\!\rangle)) && (7)\\
& && \text{by definition of continuation composition} \\
&= && ([\![\mathrm{e}_1]\!]\ (\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathrm{e}_2]\!]\ \mathrm{K}'\langle\!\langle\mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}[\mathrm{x}_1\ \mathrm{x}_2]\rangle\!\rangle))) && (8)\\
& && \text{by the induction hypothesis applied to } \mathrm{e}_2 \\
&= && ([\![\mathrm{e}_1]\!]\ (\mathbf{let}\ \mathrm{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathrm{e}_2]\!]\ \mathbf{let}\ \mathrm{x}_2 = [\cdot]\ \mathbf{in}\ \mathrm{K}'\langle\!\langle \mathrm{K}\rangle\!\rangle[\mathrm{x}_1\ \mathrm{x}_2]))) && (9)\\
& && \text{by definition of continuation composition}
\end{aligned}$$

□

Corollary 3.3. $\mathrm{K}\langle\!\langle\ [\![\mathrm{e}]\!]\ \rangle\!\rangle = [\![\mathrm{e}]\!]\ \mathrm{K}$

Next we show compositionality of the translation with respect to substitution (Lemma 3.4). While the proof relies on the previous lemma, this lemma is different in that substitution is the primary means of composition within the type system. We must essentially show that substitution is equivalent to composing via continuations. Since standard substitution does not preserve ANF, this lemma does not equate ECC^A terms, but ECC terms that have been transformed via ANF translation. We will again use language boundaries to indicate a shift from ANF to non-ANF terms. Note that this lemma relies on uniqueness of names.

Lemma 3.4 (Substitution). $[\![e[x := e']]\!]\, K \equiv \mathcal{ST}(([\![e]\!]\, K)[x := [\![e']\!]])$

Proof. By induction on the structure of e We give the key cases.

Case: $e = x$ Must show that $[\![e']\!]\, K \equiv \mathcal{ST}(([\![x]\!]\, K)[x := [\![e']\!]])$

$$\begin{aligned}
&\mathcal{ST}([\![x]\!]\, K[x := [\![e']\!]]) \\
&= \mathcal{ST}(K[x][x := [\![e']\!]]) && (10)\\
&= \mathcal{ST}(K[[\![e']\!]]) && (11)\\
&\equiv K\langle\!\langle\, [\![e']\!]\,\rangle\!\rangle && \text{by Lemma 2.5} \quad (12)\\
&\equiv [\![e']\!]\, K && \text{by Lemma 3.2} \quad (13)\\
& && (14)
\end{aligned}$$

Case: $e = \mathrm{Prop}$ Trivial.
Case: $e = \Pi\, x' : A.\, B$
Must show that $[\![\Pi\, x' : A.\, B[x := e']]\!]\, K \equiv \mathcal{ST}(([\![\Pi\, x' : A.\, B]\!]\, K)[x := [\![e']\!]])$

$$\begin{aligned}
&[\![\Pi\, x' : A.\, B[x := e']]\!]\, K \\
&= [\![\Pi\, x' : A[x := e'].\, B[x := e']]\!]\, K && (15)\\
&= K[\Pi\, x' : [\![A[x := e']]\!].\, [\![B[x := e']]\!]] && (16)\\
&\equiv K[\Pi\, x' : \mathcal{ST}([\![A]\!][x := [\![e']\!]]).\, \mathcal{ST}([\![B]\!][x := [\![e']\!]])] && \text{by the induction hypothesis} \quad (17)\\
&= \mathcal{ST}(K[\Pi\, x' : [\![A]\!].\, [\![B]\!]][x := [\![e']\!]]) && \text{by definition of substitution} \quad (18)\\
&= \mathcal{ST}(([\![\Pi\, x' : A.\, B]\!]\, K)[x := [\![e']\!]]) && \text{by definition} \quad (19)
\end{aligned}$$

Case: $e = e_1\ e_2$
Must show that $[\![(e_1\ e_2)[x := e']]\!]\, K \equiv \mathcal{ST}(([\![e_1\ e_2]\!]\, K)[x := [\![e']\!]])$

$$\begin{aligned}
&[\![(e_1\ e_2)[x := e']]\!]\, K \\
&= [\![e_1[x := e']\ e_2[x := e']]\!]\, K && \text{by substitution} \quad (20)\\
&= [\![e_1[x := e']]\!]\ \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ [\![e_2[x := e']]\!]\ \mathbf{let}\ x_2 = [\cdot]\ \mathbf{in}\ K[x_1\ x_2] && \text{by translation} \quad (21)\\
&\equiv [\![e_1[x := e']]\!]\ \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ \mathcal{ST}(([\![e_2]\!]\ \mathbf{let}\ x_2 = [\cdot]\ \mathbf{in}\ K[x_1\ x_2])[x := [\![e']\!]]) && \text{by IH applied to } e_1 \quad (22)\\
&\equiv [\![e_1]\!]\ \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ x_2 = [\cdot]\ \mathbf{in}\ K[x_1\ x_2][x := [\![e']\!]][x := [\![e']\!]] && \text{by IH applied to } e_2 \quad (23)\\
&= \mathcal{ST}(([\![e_1]\!]\ \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ x_2 = [\cdot]\ \mathbf{in}\ K[x_1\ x_2])[x := [\![e']\!]]) && \text{by substitution} \quad (24)\\
&= \mathcal{ST}(([\![e_1\ e_2]\!]\, K)[x := [\![e']\!]]) && \text{by substitution} \quad (25)
\end{aligned}$$

□

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

Lemma 3.5. *If* $\Gamma \vdash e \rhd e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$.

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

Case: $\Gamma \vdash x \rhd_\delta e'$
We must show that $[\![\Gamma]\!] \vdash [\![x]\!] \equiv [\![e']\!]$
We know that $x = e' \in \Gamma$, and by definition $x = [\![e']\!] \in [\![\Gamma]\!]$, so the goal follows by definition.

Case: $\Gamma \vdash \lambda\, x : A.\, e_1\ e_2 \triangleright_\beta e_1[x := e_2]$
We must show $[\![\Gamma]\!] \vdash [\![(\lambda\, x : A.\, e_1)\ e_2]\!] \equiv [\![e_1[x := e_2]]\!]$

$$
\begin{aligned}
& [\![\lambda\, x : A.\, e_1\ e_2]\!] \\
&= [\![\lambda\, x : A.\, e_1]\!]\ \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{x_1}\ \mathbf{x_2} && (26)\\
&= \mathbf{let}\ \mathbf{x_1} = (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!])\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{x_1}\ \mathbf{x_2} && (27)\\
&\triangleright^*\ [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!]\ \ \mathbf{x_2} && (28)\\
&= \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!])\ \mathbf{x_2}\langle\!\langle\, [\![e_2]\!]\, \rangle\!\rangle && \text{by Lemma 3.2} \quad (29)\\
&\equiv \mathrm{let}\ x_2 = [\![e_2]\!]\ \ \mathrm{in}\ (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!])\ x_2 && \text{by Lemma 2.5} \quad (30)\\
&\triangleright_\zeta\ (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!])\ \ [\![e_2]\!] && (31)\\
&\triangleright_\beta\ \mathcal{ST}([\![e_1]\!][\mathbf{x} := [\![e_2]\!]]) && (32)\\
&\equiv [\![e_1[x := e_2]]\!] && \text{by Lemma 3.4} \quad (33)
\end{aligned}
$$

□

Next we show that conversion is preserved up to equivalence. Note that past work has a minor bug in the *proof* of the following lemma [Bowman and Ahmed 2018; Bowman et al. 2018], although it does not invalidate their *theorems*. The past proofs only account for transitivity of $\triangleright^*$, 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 3.6. *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 3.5 and the induction hypothesis.
Case: [Red-Cong-Let]
We have $\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e \triangleright^* \mathrm{let}\ x = e_1\ \mathrm{in}\ e'$ and $\Gamma \vdash e \triangleright^* e'$.
We must show that $[\![\Gamma]\!] \vdash [\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e]\!] \equiv [\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e']\!]$.

$$
\begin{aligned}
& [\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e]\!] \\
&= [\![\mathrm{let}\ x = e_1\ \mathrm{in}\ y[y := e]]\!] && (34)\\
&\equiv \mathcal{ST}([\![\mathrm{let}\ x = e_1\ \mathrm{in}\ y]\!][\mathbf{y} := [\![e]\!]]) && \text{by Lemma 3.4 (Substitution)} \quad (35)\\
&\equiv \mathcal{ST}([\![\mathrm{let}\ x = e_1\ \mathrm{in}\ y]\!][\mathbf{y} := [\![e']\!]]) && \text{by the induction hypothesis} \quad (36)\\
&\equiv [\![\mathrm{let}\ x = e_1\ \mathrm{in}\ y[y := e']]\!] && \text{by Lemma 3.4} \quad (37)\\
&= [\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e']\!] && (38)
\end{aligned}
$$

□

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

Lemma 3.7. *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: [≡] Follows by Lemma 3.6.
Case: [≡-η_1]
By Lemma 3.6, we know $[\![e]\!] \equiv [\![\lambda\, x : A.\, e_1]\!]$. By transitivity, it suffices to show $[\![\lambda\, x : A.\, e_1]\!] \equiv [\![e']\!]$.
By [≡-η_1], since $[\![\lambda\, x : A.\, e_1]\!] = \boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!]$, it suffices to show that $[\![e_1]\!] \equiv [\![e']\!]\ \ x_2$

$$
\begin{aligned}
& [\![e_1]\!] \\
&\equiv [\![e'\ x_2]\!] && \text{by the induction hypothesis} \quad (39)
\end{aligned}
$$

$$= [\![e']\!]\ \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ x_1\ x_2 \tag{40}$$

$$= (\mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ x_1\ x_2)\langle\!\langle\ [\![e']\!]\ \rangle\!\rangle \qquad \text{by Lemma 3.2} \tag{41}$$

$$\equiv \mathrm{let}\ x_1 = [\![e']\!]\ \ \mathrm{in}\ x_1\ x_2 \qquad \text{by Lemma 2.5} \tag{42}$$

$$\triangleright_\zeta\ [\![e']\!]\ x_2 \tag{43}$$

Case: [$\equiv$-η_2] Essentially similar to the previous case. □

Since we implement cumulative universes through subtyping, we must also show subtyping is preserved (Lemma 3.8). The proof is completely uninteresting, except insofar as it is simple, while it seems to be impossible for CPS translation [Bowman et al. 2018]. We discuss this further in ??.

Lemma 3.8. *If* $\Gamma \vdash e \preceq e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \preceq [\![e']\!]$

Proof. By induction on the structure of $\Gamma \vdash e \preceq e'$.

Case: [$\preceq$-$\equiv$]. Follows by Lemma 3.7.
Case: [$\preceq$-Trans]. Follows the induction hypothesis.
Case: [$\preceq$-Prop]. Trivial, since $[\![\mathrm{Prop}\,]\!] = \mathbf{Prop}$ and $[\![\mathrm{Type}_{\,0}]\!] = \mathbf{Type}_{\,0}$.
Case: [$\preceq$-Cum]. Trivial, since $[\![\mathrm{Type}_{\,i}]\!] = \mathbf{Type}_{\,i}$ and $[\![\mathrm{Type}_{\,i+1}]\!] = \mathbf{Type}_{\,i+1}$.
Case: [$\preceq$-Pi].
We must show that $[\![\Gamma]\!] \vdash [\![\Pi\, x_1 : A_1.\, B_1]\!] \preceq [\![\Pi\, x_2 : A_2.\, B_2]\!]$
By definition of the translation, we must show $[\![\Gamma]\!] \vdash \mathbf{\Pi}\, \mathbf{x_1} : [\![A_1]\!].\, [\![B_1]\!] \preceq \mathbf{\Pi}\, \mathbf{x_2} : [\![A_2]\!].\, [\![B_2]\!]$.
Note that if we lifted the continuations in type annotations A_1 and A_2 outside the Π, as CBPV suggests we should, we would need a new subtyping rule that allows subtyping **let** expressions. As it is, we proceed by [$\preceq$-Pi].
It suffices to show that
(a) $[\![\Gamma]\!] \vdash [\![A_1]\!] \equiv [\![A_2]\!]$, which follows by the induction hypothesis.
(b) $[\![\Gamma]\!]\, , \mathbf{x_1} : [\![A_2]\!] \vdash [\![B_1]\!] \preceq [\![B_2]\!][\mathbf{x_2} := \mathbf{x_1}]$, which follows by the induction hypothesis.
Case: [$\preceq$-Sig]. 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 `defs`($[\![e]\!]$), expects the term `hole`($[\![e]\!]$), and has result type **B**, the translation $[\![e]\!]\ \mathbf{K}$ 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 3.11 (Type Preservation), but we need a stronger lemma first (Lemma 3.10). Since the translation is pushing computation inside-out (since continuations compose inside-out), our type-preservation lemma and proof are essentially inside-out. Instead of the expected statement, we must show that if we have a continuation **K** that expects $[\![e]\!] : [\![A]\!]$, then we get a term $[\![e]\!]\ \mathbf{K}$ of some arbitrary type **B**. In order to show that, we will have to show that $[\![e]\!] : [\![A]\!]$ and then appeal to Lemma 2.8 (Cut). Furthermore, each appeal to the inductive hypothesis will have to establish that we can in fact create well-typed continuations from the assumption that $[\![e]\!] : [\![A]\!]$.

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

We begin with a minor technical lemma (Lemma 3.9) that will come in useful in the proof of type preservation. This lemma allows us to establish that a continuation is well typed when it expects an inductively smaller translated term in its hole. It also tells us, formally, that the inductive hypothesis implies the type preservation theorem we expect.

Lemma 3.9. *If for all* $\Gamma \vdash e : A$ *and* $[\![\Gamma]\!]\, , \mathtt{defs}([\![e]\!]) \vdash \mathbf{K} : (\mathtt{hole}([\![e]\!]) : [\![A]\!]) \Rightarrow \mathbf{B}$ *we know that* $[\![\Gamma]\!] \vdash [\![e]\!]\ \mathbf{K} : \mathbf{B}$*, then* $[\![\Gamma]\!]\, , \mathtt{defs}([\![e]\!]) \vdash \mathtt{hole}([\![e]\!]) : [\![A]\!]$ *(and, incidentally,* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$*)*

Proof. Note that by Theorem 3.1 (ANF) and the definitions of `defs`($[\![e]\!]$) and `hole`($[\![e]\!]$), $[\![\Gamma]\!]\, , \mathtt{defs}([\![e]\!]) \vdash \mathtt{hole}([\![e]\!]) : [\![A]\!]$ is a sub-derivation of $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$, so it suffices to show that $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$. By the premise $[\![\Gamma]\!] \vdash [\![e]\!]\ \mathbf{K} : \mathbf{B}$, it suffices to show that $[\cdot] : (\_ : [\![A]\!]) \Rightarrow [\![A]\!]$, which is true by [K-Empty]. □

Lemma 3.10.

(1) If $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]$
(2) If $\Gamma \vdash e : A$, *and* $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash K : (\mathtt{hole}([\![e]\!]) : [\![A]\!]) \Rightarrow B$ *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: [Ax-Prop]
We must show that $[\![\Gamma]\!] \vdash [\![\mathrm{Prop}]\!]\, K : B$.
By definition of the translation, it suffices to show that $[\![\Gamma]\!] \vdash K[\mathbf{Prop}] : B$.
Note that $\mathtt{defs}([\![\mathrm{Prop}]\!]) = \cdot$; this property holds for all values.
By Lemma 2.8 (Cut), it suffices to show that
(a) $\mathtt{hole}([\![\mathrm{Prop}]\!]) = \mathbf{Prop}$, which is true by definition of the translation, and
(b) $\mathbf{Prop} : [\![\mathrm{Type}_1]\!]$, which is true by [Ax-Prop], since $[\![\mathrm{Type}_1]\!] = \mathbf{Type}_1$.
Case: [Lam]
We must show that $[\![\Gamma]\!] \vdash [\![\lambda\, x : A'.\, e']\!]\, K : B$.
That is, by definition of the translation, $[\![\Gamma]\!] \vdash K[\boldsymbol{\lambda}\, \mathbf{x} : [\![A']\!].\, [\![e']\!]] : B$.
Recall that $\mathtt{defs}([\![\lambda\, x : A'.\, e']\!]) = \cdot$, since values export no definitions.
By Lemma 2.8, it suffices to show that $[\![\Gamma]\!] \vdash \boldsymbol{\lambda}\, \mathbf{x} : [\![A']\!].\, [\![e']\!] : [\![\Pi\, x : A'.\, B']\!]$.
By definition, $[\![\Pi\, x : A'.\, B']\!] = \boldsymbol{\Pi}\, \mathbf{x} : [\![A']\!].\, [\![B']\!]$, we must show $[\![\Gamma]\!] \vdash \boldsymbol{\lambda}\, \mathbf{x} : [\![A']\!].\, [\![e']\!] : \boldsymbol{\Pi}\, \mathbf{x} : [\![A']\!].\, [\![B']\!]$.
By [Lam], it suffices to show $[\![\Gamma]\!], \mathbf{x} : [\![A']\!] \vdash [\![e']\!] : [\![B']\!]$.
Note that $[\![\Gamma]\!] \vdash [\cdot] : (\_ : [\![B']\!]) \Rightarrow [\![B']\!]$.
So, the goal follows by the induction hypothesis applied to $\Gamma, x : A' \vdash e' : B'$ with $K = [\cdot]$
Case: [If] This case is essentially similar to the **if** case for Lemma 2.12. Note that composing a continuation K with a configuration M has to perform the ANF translation for **if**.
Case: [Snd]
We must show $[\![\Gamma]\!] \vdash [\![\mathrm{snd}\, e]\!]\, K : B$, where $[\![\Gamma]\!], \mathtt{defs}([\![\mathrm{snd}\, e]\!]) \vdash K : (\mathtt{hole}([\![\mathrm{snd}\, e]\!]) : [\![B'[x := \mathrm{fst}\, e]]\!]) \Rightarrow B$ and $\Gamma \vdash e : \Sigma\, x : A'.\, B'$.
That is, by definition of the translation, we must show, $[\![\Gamma]\!] \vdash [\![e]\!]\, \mathbf{let}\, \mathbf{x}' = [\cdot]\, \mathbf{in}\, K[\mathbf{snd}\, \mathbf{x}'] : B$.
Let $K' = \mathbf{let}\, \mathbf{x}' = [\cdot]\, \mathbf{in}\, K[\mathbf{snd}\, \mathbf{x}']$.
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: $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash \mathbf{let}\, \mathbf{x}' = [\cdot]\, \mathbf{in}\, K[\mathbf{snd}\, \mathbf{x}'] : (\mathtt{hole}([\![e]\!]) : [\![\Sigma\, x : A'.\, B']\!]) \Rightarrow B$
By [K-Bind], it suffices to show that
(a) $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]) \vdash \mathtt{hole}([\![e]\!]) : [\![\Sigma\, x : A'.\, B']\!]$, which follows by Lemma 3.9 applied to the induction hypothesis for $\Gamma \vdash e : \Sigma\, x : A'.\, B'$
(b) $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), \mathbf{x}' = \mathtt{hole}([\![e]\!]) \vdash K[\mathbf{snd}\, \mathbf{x}'] : B$.
To complete this case of the proof, it suffices to show Item (b).
Note that $\mathtt{defs}([\![\mathrm{snd}\, e]\!]) = (\mathtt{defs}([\![e]\!]), \mathbf{x}' = \mathtt{hole}([\![e]\!]))$ and $\mathtt{hole}([\![\mathrm{snd}\, e]\!]) = \mathbf{snd}\, \mathbf{x}'$.
So, by Lemma 2.8 (Cut), given the type of K, it suffices to show that
$[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), \mathbf{x}' = \mathtt{hole}([\![e]\!]) \vdash \mathbf{snd}\, \mathbf{x}' : [\![B'[x := \mathrm{fst}\, e]]\!]$.
By Lemma 3.4, $[\![B'[x := \mathrm{fst}\, e]]\!] \equiv [\![B']\!][\mathbf{x} := [\![\mathrm{fst}\, e]\!]]$.
By [Conv], it suffices to show that $[\![\Gamma]\!], \mathtt{defs}([\![e]\!]), \mathbf{x}' = \mathtt{hole}([\![e]\!]) \vdash \mathbf{snd}\, \mathbf{x}' : [\![B']\!][\mathbf{x} := [\![\mathrm{fst}\, e]\!]]$.
Note that we cannot show this by the typing rule [Snd], since the substitution $[\![B']\!][\mathbf{x} := [\![\mathrm{fst}\, e]\!]]$ copies an apparently arbitrary expression $[\![\mathrm{fst}\, e]\!]$ into the type, instead of the expected sub-expression $\mathbf{fst}\, \mathbf{x}'$. That is, [Snd] tells us $\mathbf{snd}\, \mathbf{x}' : [\![B']\!][\mathbf{x} := \mathbf{fst}\, \mathbf{x}']$ but we must show $\mathbf{snd}\, \mathbf{x}' : [\![B']\!][\mathbf{x} := [\![\mathrm{fst}\, e]\!]]$. The translation has disrupted the dependency on e, changing the type that depended on the specific value e into a type that depends on an apparently arbitrary value $\mathbf{x}'$. 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 $[\![\mathsf{fst}\ e]\!] \equiv \mathbf{fst}\ x'$, under the definitions we have accumulated from continuation typing. This follows by Corollary 2.11.

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

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

Case: [App]

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

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

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

By the inductive hypothesis applied to $\Gamma \vdash e_1 : B'[x := e_1]$, it suffices to show that
$[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]) \vdash \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ x_2 = [\cdot]\ \mathbf{in}\ K[x_1\ x_2] : (\mathtt{hole}([\![e_1]\!]) : [\![\Pi\, x : A'.\ B']\!]) \Rightarrow B$

To show this, by [K-Bind], it suffices to show

(a) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]) \vdash \mathtt{hole}([\![e_1]\!]) : [\![\Pi\, x : A'.\ B']\!]$, which follows by Lemma 3.9 applied to the induction hypothesis for $\Gamma \vdash e_1 : \Pi\, x : A'.\ B'$

(b) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]) \vdash [\![e_2]\!]\ \mathbf{let}\ x_2 = [\cdot]\ \mathbf{in}\ K[x_1\ x_2] : B$

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

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

By Lemma 2.8 (Cut), we must show
$[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]), x_2 = \mathtt{hole}([\![e_2]\!]) \vdash x_1\ x_2 : [\![B'[x := e_2]]\!]$.

By Lemma 3.4 and [Conv], it suffices to show
$[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]), x_2 = \mathtt{hole}([\![e_2]\!]) \vdash x_1\ x_2 : [\![B']\!][x := [\![e_2]\!]]$.

As in the proof case for [Snd], we cannot proceed directly by [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 [Bowman et al. 2018]. But, again, we know by Corollary 2.11 that under these exported definitions, $[\![e_2]\!] \equiv x_2$. So by [Conv], it suffices to show
$[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]), x_2 = \mathtt{hole}([\![e_2]\!]) \vdash x_1\ x_2 : [\![B']\!][x := x_2]$.

By [App] it suffices to show

(a) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]), x_2 = \mathtt{hole}([\![e_2]\!]) \vdash x_1 : \Pi\, x : [\![A']\!].\ [\![B']\!]$, which follows by Lemma 3.9 applied to the induction hypothesis for $\Gamma \vdash e_1 : \Pi\, x : A'.\ B'$

(b) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]), x_2 = \mathtt{hole}([\![e_2]\!]) \vdash x_2 : [\![A']\!]$, which follows by Lemma 3.9 applied to the induction hypothesis for $\Gamma \vdash e_2 : A'$.

Case: [Let]

We must show that $[\![\Gamma]\!] \vdash [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\ K : B$.

That is, by definition, $[\![\Gamma]\!] \vdash [\![e_1]\!]\ \mathbf{let}\ x_1 = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ K : B$.

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

By [K-Bind], it suffices to show

(a) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]) \vdash \mathtt{hole}([\![e_1]\!]) : [\![[\![A]\!]]\!]$, which follows by Lemma 3.9 applied to the induction hypothesis for $\Gamma \vdash e_1 : A$,

(b) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]) \vdash [\![e_2]\!]\ K : B$.

Item (b) follows from the induction hypothesis applied to $\Gamma, x = e_1 \vdash e_2 : B'$ 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:
$[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]) \vdash K : (\mathtt{hole}([\![e_2]\!]) : [\![B'[x_1 := e_1]]\!]) \Rightarrow B$

Currently, we know by our premises that
$[\![\Gamma]\!], \mathtt{defs}([\![\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2]\!]) \vdash K : (\mathtt{hole}([\![\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2]\!]) : [\![B'[x_1 := e_1]]\!]) \Rightarrow B$

So it suffices to show that

(a) $\mathtt{defs}([\![\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2]\!]) = (\mathtt{defs}([\![e_1]\!]), x_1 = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]))$

(b) $\mathtt{hole}([\![\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2]\!]) = \mathtt{hole}([\![e_2]\!])$

both of which are straightforward by definition. □

$\boxed{\mathsf{v} \approx \mathbf{V}}$

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

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

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

Fig. 16. Separate Compilation Definitions

Theorem 3.11 (Type Preservation). *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]$

Proof. By Lemma 3.10, it suffices to show that $[\![\Gamma]\!] \vdash [\cdot] : (\_ : [\![\mathsf{A}]\!]) \Rightarrow [\![\mathsf{A}]\!]$, which is trivial. □

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

We first define observations, and when observations are related across languages. Without such a relation, the best we can prove is that the *translation* of the value v produced in the source is *definitionally equivalent* to the value we get by running the translated term, *i.e.*, we would get $[\![\mathsf{v}]\!] \equiv \mathbf{eval}([\![\mathsf{e}]\!])$. This fails to tell us how $[\![\mathsf{v}]\!]$ 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 $\mathsf{v} \approx \mathbf{V}$ to compare ground values in Figure 16.

We define linking as substitution with well-typed closed terms, and define a closing substitution γ with respect to the environment Γ (also in Figure 16). Linking is defined by closing a term e such that $\Gamma \vdash \mathsf{e} : \mathsf{A}$ with a substitution $\Gamma \vdash \gamma$, written $\gamma(\mathsf{e})$. Any γ is valid for Γ if it maps each $\mathsf{x} : \mathsf{A} \in \Gamma$ to a closed term e of type A. For definitions in Γ, we require that if $\mathsf{x} = \mathsf{e} \in \Gamma$, then $\gamma[\mathsf{x} \mapsto \gamma(\mathsf{e})]$, 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 3.12 (Correctness of Separate Compilation). *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$, *(and* A *ground) and* $\Gamma \vdash \gamma$ *then* $\mathbf{eval}([\![\gamma]\!]\,([\![\mathsf{e}]\!])) \approx \mathsf{eval}(\gamma(\mathsf{e}))$.

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.

$$\begin{tikzcd}
\mathsf{eval}(\gamma(\mathsf{e})) \arrow[r, "\equiv"] \arrow[d, "\equiv"] & {[\![\gamma(\mathsf{e})]\!]} \arrow[d, "\equiv"] \\
\mathbf{eval}([\![\gamma([\![\mathsf{e}]\!])]\!]) \arrow[r, "\equiv"] & {[\![\gamma([\![\mathsf{e}]\!])]\!]}
\end{tikzcd}$$

□

4 JOIN-POINT OPTIMIZATION

Recall from Figure 10 that the composition of a continuation K with a configuration $\text{if } V \text{ then } M_1 \text{ else } M_2$, $K\langle\langle \text{if } V \text{ then } M_1 \text{ else } M_2 \rangle\rangle$, duplicates K in the branches: intuitively, $\text{if } V \text{ then } K\langle\langle M_1\rangle\rangle \text{ else } K\langle\langle M_2\rangle\rangle$, although for type checking we need the ANF version of $K\langle\langle \text{subst}_{V=\text{true}}\ M_1\rangle\rangle$ for the branches. Similarly, the ANF translation in Figure 15 performs the same duplication when translating **if** expressions. This can cause exponential code duplication, which is no problem in theory but is a problem in practice.

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

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

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

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

From β and ζ reduction, it is simple to conclude the desired equivalence, formally stated below as Lemma 4.1.

Lemma 4.1 (Join Point Equivalence).

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

Proof. By β and ζ reduction, it suffices to show that
$K\langle\langle \text{if } V \text{ then } M_1 \text{ else } M_2\rangle\rangle \equiv \text{if } V \text{ then } (K[\text{subst}_{V=\text{true}}\ x][M_1//x]) \text{ else } (K[\text{subst}_{V=\text{false}}\ x][M_2//x])$
which holds by definition.
Note that since equivalence is defined over untyped syntax, the annotation $B[x := V]$ is irrelevant. □

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

Lemma 4.2 (Join Point Well Typed). *If* $\Gamma \vdash K\langle\langle \text{if } V \text{ then } M_1 \text{ else } M_2\rangle\rangle : C$, $\Gamma \vdash \text{if } V \text{ then } M_1 \text{ else } M_2 : B[x := V]$, *and* $\Gamma \vdash K : \texttt{hole}(\text{if } V \text{ then } M_1 \text{ else } M_2) : B' \Rightarrow C$ *then*

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

Proof. By definition, $\Gamma \vdash K\langle\langle \text{if } V \text{ then } M_1 \text{ else } M_2\rangle\rangle : C$ implies
$\Gamma \vdash \text{if } V \text{ then } K[\text{subst}_{V=\text{true}}\ x][M_1//x] \text{ else } K[\text{subst}_{V=\text{false}}\ x][M_2//x] : C$
Let $\Gamma' = (\Gamma, f = \lambda\, x : (B[x := V]).\ K[x])$, and note that $f : (B[x := V]) \rightarrow C$.
By [If] and Lemma 2.12 (Heterogeneous Cut), it suffices to show that

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

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

It's a simple corollary to show we can now modify the ANF translation from Section 3 to use the definition in Figure 17, and it is still correct and type preserving. However, note that the new translation requires access to the

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

$$\vdots$$

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

Fig. 17. Join-Point Optimized ANF Translation

type B from the derivation. We can do this either by defining the translation by induction over typing derivations, or (preferably) modifying the syntax of if to include B as a type annotation, similar to dependent pairs.

REFERENCES

Gilles Barthe, John Hatcliff, and Morten Heine B. Sørensen. 1999. CPS Translations and Applications: The Cube and Beyond. *Higher-Order and Symbolic Computation* 12, 2 (Sept. 1999). https://doi.org/10.1023/a:1010000206149

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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