Typed Closure Conversion for the Calculus of Constructions (Technical Appendix)

WILLIAM J. BOWMAN, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

PREFACE

This is an expanded version of the technical sections for the paper by the same title. This contains extended versions of figured and proofs, and additional discussions and explanations.

1 SOURCE: CALCULUS OF CONSTRUCTIONS (CC)

Our source language is a variant of the Calculus of Constructions (CC) extended with strong dependent pairs (Σ types) and η-equivalence for functions, which we typeset in a non-bold, blue, sans-serif font. This model is based on the CIC specification used in Coq [5, Chapter 4]. For brevity, we omit base types from this formal system but will freely use base types like natural numbers in examples.

We present the syntax of CC in Figure 1. Universes, or sorts, U are essentially the types of types. CC includes one impredicative universe $\star$, and one predicative universe $\square$. Expressions have no explicit distinction between terms, types, or kinds, but we usually use the meta-variable e to evoke a term expression and A or B to evoke a type expression. Expressions include names x, the universe $\star$, functions $\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}$, application $\mathsf{e}_1\ \mathsf{e}_2$, dependent function types $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}$, dependent let $\mathsf{let}\ \mathsf{x} = \mathsf{e} : \mathsf{A}\ \mathsf{in}\ \mathsf{e}'$, Σ types $\Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}$, dependent pairs $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle\ \mathsf{as}\ \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}$, first projections $\mathsf{fst}\ \mathsf{e}$ and second projections $\mathsf{snd}\ \mathsf{e}$. The universe $\square$ is only used by the type system and is not a valid term. As syntactic sugar, we omit the type annotations on dependent let $\mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}'$ and on dependent pairs $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ when they are irrelevant or obvious from context. We also write function types as $\mathsf{A} \longrightarrow \mathsf{B}$ when the result B does not depend on the argument. Environments Γ include assumptions $\mathsf{x} : \mathsf{A}$ that a name x has type A, and definitions $\mathsf{x} = \mathsf{e} : \mathsf{A}$ that name x refers to e of type A.

We define conversion, or reduction, and definitional equivalence for CC in Figure 2. Conversion here is defined for deciding equivalence between types (which include terms), but it can also be viewed as the operational semantics of CC terms. The small-step reduction $\Gamma \vdash \mathsf{e} \rhd \mathsf{e}'$ reduces the expression e to the term e' under the local environment Γ, which we usually leave implicit for brevity. The local environment is necessary to convert a name to its definition. Each conversion rule is labeled, and when we refer to conversion with an unlabeled arrow $\mathsf{e} \rhd \mathsf{e}'$, we mean that e reduces to e' by *some* reduction rule, *i.e.*, either $\rhd_\delta$, $\rhd_\zeta$, $\rhd_\beta$, $\rhd_{\pi_1}$, or $\rhd_{\pi_2}$. We write $\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'$ to mean the reflexive, transitive, contextual closure of the relation $\Gamma \vdash \mathsf{e} \rhd \mathsf{e}'$. Essentially, $\mathsf{e} \rhd^* \mathsf{e}'$ runs e using the $\rhd$ relation any number of times, under any arbitrary context. The $\rhd^*$ relation introduces a definition into the local environment when descending into the body of a dependent let. That is, we have the following closure rule for $\rhd^*$.

$$\frac{\Gamma, \mathsf{x} = \mathsf{e} \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_2}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}_1 \rhd^* \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}_2}$$

We define equivalence $\Gamma \vdash \mathsf{e} \equiv \mathsf{e}'$ as reduction in the $\rhd^*$ relation up to η-equivalence, as in Coq [5, Chapter 4].

In Figure 3, we present the typing rules. The type system is standard.

Functions $\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}$ have dependent function type $\Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}$ ([Lam]). The dependent function type describes that the function takes an argument, x, of type A, and returns something of type B where B may refer to, *i.e.*, *depends on*, the value of the argument x. We can use this to write polymorphic functions, such as the polymorphic identity function described by the type $\Pi\, \mathsf{A} : \star.\, \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{A}$, or functions with pre/post conditions, such as the division function described by $\Pi\, \mathsf{x} : \mathsf{Nat}.\, \Pi\, \mathsf{y} : \mathsf{Nat}.\, \Pi\, \_ : \mathsf{y} > 0.\, \mathsf{Nat}$, which statically ensures that we never divide by zero by requiring a proof that its second argument is greater than zero.

Applications $\mathsf{e}_1\ \mathsf{e}_2$ have type $\mathsf{B}[\mathsf{e}_2/\mathsf{x}]$ ([App]), *i.e.*, the result type B of the function e_1 with the argument e_2 substituted for the name of the argument x. Using this rule and our example of the division function $\mathsf{div} : \Pi\, \mathsf{x} : \mathsf{Nat}.\, \Pi\, \mathsf{y} : \mathsf{Nat}.\, \Pi\, \_ :$

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

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

Fig. 1. CC Syntax

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

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

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

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

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

Fig. 2. CC Conversion and Equivalence

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

$$\dfrac{\vdash \Gamma}{\Gamma \vdash \star : \square}\ [\textsc{Ax-}*] \qquad \dfrac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ [\textsc{Var}] \qquad \dfrac{\Gamma \vdash e : A \qquad \Gamma, x = e : A \vdash e' : B}{\Gamma \vdash \mathsf{let}\ x = e : A\ \mathsf{in}\ e' : B[e/x]}\ [\textsc{Let}]$$

$$\dfrac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi\, x : A.\, B : \star}\ [\textsc{Prod-}*] \qquad \dfrac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi\, x : A.\, B : \square}\ [\textsc{Prod-}\square] \qquad \dfrac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B}\ [\textsc{Lam}]$$

$$\dfrac{\Gamma \vdash e : \Pi\, x : A'.\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\ e' : B[e'/x]}\ [\textsc{App}] \qquad \dfrac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma\, x : A.\, B : \star}\ [\textsc{Sig-}*] \qquad \dfrac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Sigma\, x : A.\, B : \square}\ [\textsc{Sig-}\square]$$

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

Fig. 3. CC Typing

$y > 0.\ \mathsf{Nat}$, we type check the term $\mathsf{div}\ 4\ 2 : \Pi\ \_ : 2 > 0.\ \mathsf{Nat}$. Notice that the term variable y in the type has been replaced with the value of the argument 2.

Dependent pairs $\langle e_1, e_2 \rangle$ have type $\Sigma\, x : A.\, B$ ([Pair]). Again, this type is a binding form. The type B of the second component of the pair can refer to the first component of the pair by the name x. We see in the rule [Snd] that the type of $\mathsf{snd}\ e$ is $B[\mathsf{fst}\ e/x]$, *i.e.*, the type B of the second component of the pair with the name x substituted by $\mathsf{fst}\ e$. We can use this to encode refinement types, such as the describing positive numbers by $\Sigma\, x : \mathsf{Nat}.\ x > 0$, *i.e.*, a pair of a number x with a proof that x is greater than 0.

$\boxed{\vdash \Gamma}$

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

Fig. 4. CC Well-Formed Environments

Since types are also terms, we have typing rules for types. The type of $\star$ is $\square$. We call $\star$ the universe of small types and $\square$ the universe of large types. Intuitively, small types are the types of programs while large types are the types of types and type-level computations. Since no user can write down $\square$, we need not worry about the type of $\square$. In [Prod-*], we assign the type $\star$ to the dependent function type when the result type is also $\star$. This rule allows *impredicative* functions, since it allows forming a function that quantifies over large types but is in the universe of small types. The rule [Prod-$\square$] looks similar, but is implicitly predicative, since there is no universe larger than $\square$ to quantify over. (We could combine the rules for Π, but explicit separation helps clarify the issue of predicativity when compared with the rules for Σ types, which cannot by combined.) Formation rules for Σ types have an important restriction: it is unsound to allow impredicativity in strong dependent pairs [2, 3]. The [Sig-*] rule only allows quantifying over a small type when forming a small dependent pair. The [Sig-$\square$] rule allows quantifying over either small or large types when forming a large Σ. As usual in models of dependent type theory, we exclude base types, although they are simple to add.

The rule [Conv] allows resolving type equivalence and reducing terms in types. For instance, if we want to show that $e : \Sigma\, x : \mathsf{Nat}.\, x = 2$ but we have $e : \Sigma\, x : \mathsf{Nat}.\, x = 1 + 1$, the [Conv] rule performs this reduction. Note while our equivalence relation is untyped, the [Conv] rule ensures that A and B are well-typed before appealing to equivalence, ensuring decidability. (It is a standard lemma that if $\Gamma \vdash e : A$, then $\Gamma \vdash A : U$ [4].)

Finally, we extend well-typedness to well-formedness of environments $\vdash \Gamma$ in Figure 4.

$$\begin{array}{lrcl}
\textit{Universes} & \mathbf{U} & ::= & \star \mid \square \\
\textit{Expressions} & \mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \star \mid \mathbf{let}\, \mathbf{x} = \mathbf{e} : \mathbf{A}\, \mathbf{in}\, \mathbf{e} \mid \mathbf{1} \mid \langle\rangle \mid \mathbf{Code}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B} \mid \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e} \\
 & & \mid & \Pi\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \mid \langle\!\langle \mathbf{e}, \mathbf{e} \rangle\!\rangle \mid \mathbf{e}\ \mathbf{e}' \mid \Sigma\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \mid \mathbf{fst}\, \mathbf{e} \mid \mathbf{snd}\, \mathbf{e}
\end{array}$$

Fig. 5. CC-CC Syntax

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

$$\begin{array}{rcll}
\mathbf{x} & \rhd_{\delta} & \mathbf{e} & \text{where } \mathbf{x} = \mathbf{e} : \mathbf{A} \in \Gamma \\
\mathbf{let}\, \mathbf{x} = \mathbf{e} : \mathbf{A}\, \mathbf{in}\, \mathbf{e}_1 & \rhd_{\zeta} & \mathbf{e}_1[\mathbf{e}/\mathbf{x}] & \\
\langle\!\langle \boldsymbol{\lambda}\, \mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}.\, \mathbf{e}_1, \mathbf{e}' \rangle\!\rangle\ \mathbf{e} & \rhd_{\beta} & \mathbf{e}_1[\mathbf{e}'/\mathbf{x}'][\mathbf{e}/\mathbf{x}] & \\
\mathbf{fst}\, \langle \mathbf{e}_1, \mathbf{e}_2 \rangle & \rhd_{\pi_1} & \mathbf{e}_1 & \\
\mathbf{snd}\, \langle \mathbf{e}_1, \mathbf{e}_2 \rangle & \rhd_{\pi_2} & \mathbf{e}_2 &
\end{array}$$

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

$$\frac{\Gamma \vdash \mathbf{e}_1 \rhd^* \mathbf{e} \qquad \Gamma \vdash \mathbf{e}_2 \rhd^* \mathbf{e}}{\Gamma \vdash \mathbf{e}_1 \equiv \mathbf{e}_2}\ [\equiv]$$

$$\frac{\Gamma \vdash \mathbf{e}_1 \rhd^* \langle\!\langle \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_1', \mathbf{e}' \rangle\!\rangle \qquad \Gamma \vdash \mathbf{e}_2 \rhd^* \mathbf{e}_2' \qquad \Gamma, \mathbf{x} : \mathbf{A} \vdash \mathbf{e}_1[\mathbf{e}'/\mathbf{x}'] \equiv \mathbf{e}_2'\ \mathbf{x}}{\Gamma \vdash \mathbf{e}_1 \equiv \mathbf{e}_2}\ [\equiv\text{-}\textsc{Clo}_1]$$

$$\frac{\Gamma \vdash \mathbf{e}_2 \rhd^* \langle\!\langle \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_2', \mathbf{e}' \rangle\!\rangle \qquad \Gamma \vdash \mathbf{e}_1 \rhd^* \mathbf{e}_1' \qquad \Gamma, \mathbf{x} : \mathbf{A} \vdash \mathbf{e}_1'\ \mathbf{x} \equiv \mathbf{e}_2'[\mathbf{e}'/\mathbf{x}']}{\Gamma \vdash \mathbf{e}_1 \equiv \mathbf{e}_2}\ [\equiv\text{-}\textsc{Clo}_2]$$

Fig. 6. CC-CC Conversion and Equivalence

2 TARGET: CC, CLOSURE-CONVERTED (CC-CC)

The target language CC-CC is based on CC, but first-class functions are replaced by closed code and closures. We add a primitive unit type $\mathbf{1}$ to support encoding environments. This language is typeset in a **bold, red, serif font**.We extend the syntax of expressions, Figure 5, with a unit value $\langle\rangle$ and its type $\mathbf{1}$, closed code $\boldsymbol{\lambda}\, \mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}.\, \mathbf{e}$ and dependent code types $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B}$, and closure values $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ and dependent closure types $\Pi\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}$. The syntax of application $\mathbf{e}\ \mathbf{e}'$ is unchanged, but it now applies closures instead of functions.

We define additional syntactic sugar for sequences of terms, to support writing environments whose length is arbitrary. We write a sequence of terms $\mathbf{e}_i \ldots$ to mean a sequence of length $|i|$ of expressions $\mathbf{e}_{i_0}, \ldots, \mathbf{e}_{i_n}$. We extend the notation to patterns such as $\mathbf{x}_i : \mathbf{A}_i \ldots$, which implies two sequences $\mathbf{x}_{i_0}, \ldots, \mathbf{x}_{i_n}$ and $\mathbf{A}_0, \ldots, \mathbf{A}_{i_n}$ each of length $|i|$. We define environments as dependent n-tuples, written $\langle \mathbf{e}_i \ldots \rangle\ \mathbf{as}\ \Sigma\,(\mathbf{x}_i : \mathbf{A}_i \ldots)$. We encode dependent n-tuples as nested dependent pairs followed by a unit value, *i.e.*, $\langle \mathbf{e}_0, \langle \ldots, \langle \mathbf{e}_i, \langle\rangle \rangle\rangle\rangle$. We omit the annotation on n-tuples $\langle \mathbf{e}_i \ldots \rangle$ when it is obvious from context. We also define pattern matching on n-tuples, written $\mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{e}'\, \mathbf{in}\, \mathbf{e}$, to perform the necessary nested projections, *i.e.*, $\mathbf{let}\, \mathbf{x}_0 = \mathbf{fst}\, \mathbf{e}'\, \mathbf{in}\, \ldots\, \mathbf{let}\, \mathbf{x}_i = \mathbf{fst}\, \mathbf{snd}\, \ldots\, \mathbf{snd}\, \mathbf{e}'\, \mathbf{in}\, \mathbf{e}$.

In Figure 6 we present the additional conversion and equivalence rules for CC-CC. Code cannot be applied directly, but must be part of a closure. Closures applied to an argument β-reduce, applying the underlying code to the environment and the argument. All the other conversion rules remain unchanged. For equivalence, we no longer have the usual η rules, since functions have been turned into closures. Instead, we need η rules for closures.

We give the typing rules in Figure 7. Most rules are unchanged from the source language. The most interesting rule is [Code], which that code only type checks when it is closed. This rule captures the entire point of typed closure conversion and gives us static machine-checked guarantees that our translation produces closed code. The typing rule [Clo] for closures $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ substitutes the environment $\mathbf{e}'$ into the type of the closure. This is similar to the CC rule [App] that substitutes a function argument into the result type of a function. This is also critical to type

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \star : \square}\;[\textsc{Ax-*}] \qquad \frac{x : A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A}\;[\textsc{Var}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash 1 : \star}\;[\textsc{T-Unit}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \langle\rangle : 1}\;[\textsc{Unit}]$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x = e : A \vdash e' : B}{\Gamma \vdash \mathrm{let}\, x = e : A \;\mathrm{in}\; e' : B[e/x]}\;[\textsc{Let}] \qquad \frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi\, x : A.\, B : \star}\;[\textsc{Prod-*}]$$

$$\frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi\, x : A.\, B : \square}\;[\textsc{Prod-}\square] \qquad \frac{\Gamma, x' : A', x : A \vdash B : \star}{\Gamma \vdash \mathrm{Code}\,(x' : A', x : A).\, B : \star}\;[\textsc{T-Code-*}]$$

$$\frac{\Gamma, x' : A', x : A \vdash B : \square}{\Gamma \vdash \mathrm{Code}\,(x : A, x' : A').\, B : \square}\;[\textsc{T-Code-}\square] \qquad \frac{\cdot, x' : A', x : A \vdash e : B}{\Gamma \vdash \lambda\,(x' : A', x : A).\, e : \mathrm{Code}\,(x' : A', x : A).\, B}\;[\textsc{Code}]$$

$$\frac{\Gamma \vdash e : \mathrm{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash \langle\!\langle e, e' \rangle\!\rangle : \Pi\, x : A[e'/x'].\, B[e'/x']}\;[\textsc{Clo}] \qquad \frac{\Gamma \vdash e : \Pi\, x : A'.\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\; e' : B[e'/x]}\;[\textsc{App}]$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma\, x : A.\, B : \star}\;[\textsc{Sig-*}] \qquad \frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : \square}{\Gamma \vdash \Sigma\, x : A.\, B : \square}\;[\textsc{Sig-}\square] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathrm{fst}\, e : A}\;[\textsc{Fst}]$$

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

Fig. 7. CC-CC Typing

preservation, since our translation must generate closure types with free variables and then synchronize the closure type containing free variables with a closed code type. As with Π types in CC, we have two rules for well typed Code types. The rule [T-Code-*] allows impredicativity in ⋆, while [T-Code-□] is predicative.

2.1 Type Safety and Consistency

We prove that CC-CC is type safe when interpreted as a programming language and consistent when interpreted as a logic. Type safety guarantees that all programs in CC-CC have well-defined behavior, and consistency ensures that when interpreting types as propositions and programs as proofs, we cannot prove False in CC-CC. We prove both theorems by giving a model of CC-CC in CC, *i.e.*, by encoding the target language in the source language. The model reduces type safety and consistency of CC-CC to that of CC, which is known to be type safe and consistent. This standard technique is well explained by Boulier et al. [1].

We construct a model essentially by "decompiling" closures, translating code to functions and closures to partial application. To show this translation is a model, we need to show that it preserves falseness—*i.e.*, that we translate False to False—and show that the translation is type-preserving—*i.e.*, we translate any well-typed CC-CC program (valid proof) into a well-typed program in CC. To extend the model to type safety, we must also show that the translation preserves reduction semantics—*i.e.*, that reducing an expression in CC-CC is essentially equivalent to reducing the translated term in CC. Since our type system includes reduction, we already prove this to show type preservation.

We then prove consistency and type safety of CC-CC by contradiction. If CC-CC were inconsistent, then we could prove the proposition False in CC-CC, and translate that proof into a valid proof of False in CC. But since CC is consistent, we can never produce a proof of False in CC, therefore we could not have constructed one in CC-CC. A similar argument applies for type safety. Since we preserve reduction semantics in CC-CC, if a term had undefined behavior, we could translate the term into a CC term with undefined behavior. However, CC has no terms with undefined behavior, hence neither does CC-CC.

The translation from CC-CC to CC, Figure 8, is defined on typing derivations. We use the following notation.

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

$$\frac{}{\Gamma \vdash \star : \square \rightsquigarrow_{\circ} \star}\ [\text{M-*}] \qquad \frac{}{\Gamma \vdash \mathrm{x} : \mathrm{A} \rightsquigarrow_{\circ} \mathrm{x}}\ [\text{M-Var}] \qquad \frac{}{\Gamma \vdash 1 : \star \rightsquigarrow_{\circ} \Pi\, \alpha : \star.\ \Pi\, \mathrm{x} : \alpha.\ \alpha}\ [\text{M-T-Unit}]$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1 \rightsquigarrow_{\circ} \lambda\, \alpha : \star.\ \lambda\, \mathrm{x} : \alpha.\ \mathrm{x}}\ [\text{M-Unit}] \qquad \frac{\Gamma \vdash \mathrm{e} : \mathrm{A} \rightsquigarrow_{\circ} \mathrm{e} \qquad \Gamma, \mathrm{x} = \mathrm{e} : \mathrm{A} \vdash \mathrm{e}' : \mathrm{B} \rightsquigarrow_{\circ}}{\Gamma \vdash \mathbf{let}\ \mathrm{x} = \mathrm{e} : \mathrm{A}\ \mathbf{in}\ \mathrm{e}' : \mathrm{B}[\mathrm{e}/\mathrm{x}] \rightsquigarrow_{\circ} \mathsf{let}\ \mathrm{x} = \mathrm{e} : \mathrm{A}\ \mathsf{in}\ \mathrm{e}'}\ [\text{M-Let}]$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \star \rightsquigarrow_{\circ} \mathrm{B}}{\Gamma \vdash \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} : \star \rightsquigarrow_{\circ} \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B}}\ [\text{M-Prod-*}] \qquad \frac{\Gamma \vdash \mathrm{A} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \square \rightsquigarrow_{\circ} \mathrm{B}}{\Gamma \vdash \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} : \square \rightsquigarrow_{\circ} \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B}}\ [\text{M-Prod-}\square]$$

$$\frac{\Gamma \vdash \mathrm{A}' : \mathrm{U}' \rightsquigarrow_{\circ} \mathrm{A}' \qquad \Gamma, \mathrm{x}' : \mathrm{A}' \vdash \mathrm{A} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \star \rightsquigarrow_{\circ} \mathrm{B}}{\Gamma \vdash \mathbf{Code}\,(\mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A}).\ \mathrm{B} : \star \rightsquigarrow_{\circ} \Pi\, \mathrm{x}' : \mathrm{A}'.\ \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B}}\ [\text{M-T-Code-*}]$$

$$\frac{\Gamma \vdash \mathrm{A}' : \mathrm{U}' \rightsquigarrow_{\circ} \mathrm{A}' \qquad \Gamma, \mathrm{x}' : \mathrm{A}' \vdash \mathrm{A} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \square \rightsquigarrow_{\circ} \mathrm{B}}{\Gamma \vdash \mathbf{Code}\,(\mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A}).\ \mathrm{B} : \square \rightsquigarrow_{\circ} \Pi\, \mathrm{x}' : \mathrm{A}'.\ \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B}}\ [\text{M-T-Code-}\square]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{A}' : \mathrm{U}' \rightsquigarrow_{\circ} \mathrm{A}' \\ \Gamma, \mathrm{x}' : \mathrm{A}' \vdash \mathrm{A} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{B} \qquad \Gamma, \mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A} \vdash \mathrm{e} : \mathrm{B} \rightsquigarrow_{\circ} \mathrm{e}\end{array}}{\Gamma \vdash \lambda\,(\mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A}).\ \mathrm{e} : \mathbf{Code}\,(\mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A}).\ \mathrm{B} \rightsquigarrow_{\circ} \lambda\, \mathrm{x}' : \mathrm{A}'.\ \lambda\, \mathrm{x} : \mathrm{A}.\ \mathrm{e}}\ [\text{M-Code}]$$

$$\frac{\Gamma \vdash \mathrm{e} : \mathbf{Code}\,(\mathrm{x}' : \mathrm{A}', \mathrm{x} : \mathrm{A}).\ \mathrm{B} \rightsquigarrow_{\circ} \mathrm{e} \qquad \Gamma \vdash \mathrm{e}' : \mathrm{A}' \rightsquigarrow_{\circ} \mathrm{e}'}{\Gamma \vdash \langle\!\langle \mathrm{e}, \mathrm{e}' \rangle\!\rangle : \Pi\, \mathrm{x} : \mathrm{A}[\mathrm{e}'/\mathrm{x}].\ \mathrm{B}[\mathrm{e}'/\mathrm{x}] \rightsquigarrow_{\circ} \mathrm{e}\ \mathrm{e}'}\ [\text{M-Clo}]$$

$$\frac{\Gamma \vdash \mathrm{e} : \Pi\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} \rightsquigarrow_{\circ} \mathrm{e} \qquad \Gamma \vdash \mathrm{e}' : \mathrm{A} \rightsquigarrow_{\circ} \mathrm{e}'}{\Gamma \vdash \mathrm{e}\ \mathrm{e}' : \mathrm{B}[\mathrm{e}'/\mathrm{x}] \rightsquigarrow_{\circ} \mathrm{e}\ \mathrm{e}'}\ [\text{M-App}] \qquad \frac{\Gamma \vdash \mathrm{A} : \star \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \star \rightsquigarrow_{\circ} \mathrm{B}}{\Gamma \vdash \Sigma\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} : \star \rightsquigarrow_{\circ} \Sigma\, \mathrm{x} : \mathrm{A}.\ \mathrm{B}}\ [\text{M-Sig-*}]$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{U} \rightsquigarrow_{\circ} \mathrm{A} \qquad \Gamma, \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \square \rightsquigarrow_{\circ} \mathrm{B}}{\Gamma \vdash \Sigma\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} : \square \rightsquigarrow_{\circ} \Sigma\, \mathrm{x} : \mathrm{A}.\ \mathrm{B}}\ [\text{M-Sig-}\square] \qquad \frac{\Gamma \vdash \mathrm{e} : \Sigma\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} \rightsquigarrow_{\circ} \mathrm{e}}{\Gamma \vdash \mathbf{fst}\ \mathrm{e} : \mathrm{A} \rightsquigarrow_{\circ} \mathsf{fst}\ \mathrm{e}}\ [\text{M-Fst}]$$

$$\frac{\Gamma \vdash \mathrm{e} : \Sigma\, \mathrm{x} : \mathrm{A}.\ \mathrm{B} \rightsquigarrow_{\circ} \mathrm{e}}{\Gamma \vdash \mathbf{snd}\ \mathrm{e} : \mathrm{A} \rightsquigarrow_{\circ} \mathsf{snd}\ \mathrm{e}}\ [\text{M-Snd}] \qquad \frac{\Gamma \vdash \mathrm{e} : \mathrm{B} \rightsquigarrow_{\circ} \mathrm{e}}{\Gamma \vdash \mathrm{e} : \mathrm{A} \rightsquigarrow_{\circ} \mathrm{e}}\ [\text{M-Conv}]$$

Fig. 8. Translation from CC-CC to CC

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

The CC expression e° refers to the expression produced by translating the CC-CC expression e, with the typing derivation for e as an implicit argument.

The rule [M-Code] translates a code type $\mathbf{Code}\,(\mathrm{n} : \mathrm{A}', \mathrm{x} : \mathrm{A}).\ \mathrm{B}$ to the curried function type $\Pi\, \mathrm{n} : \mathrm{A}'^{\circ}.\ \Pi\, \mathrm{x} : \mathrm{A}^{\circ}.\ \mathrm{B}^{\circ}$. The rule [M-Code] models code $\lambda\, \mathrm{n} : \mathrm{A}', \mathrm{x} : \mathrm{A}.\ \mathrm{e}$ as a curried function $\lambda\, \mathrm{n} : \mathrm{A}'^{\circ}.\ \lambda\, \mathrm{x} : \mathrm{A}^{\circ}.\ \mathrm{e}^{\circ}$. Observe that the inner function produced in CC is not closed, but that is not a problem since the model only exists to prove type safety and consistency. It is only in CC-CC programs that code must be closed. The rule [M-Clo] models a closure $\langle\!\langle \mathrm{e}, \mathrm{e}' \rangle\!\rangle$ as the application $\mathrm{e}^{\circ}\ \mathrm{e}'^{\circ}$—*i.e.*, the application of the function e° to its environment e'°. We model Unit with the standard Church encoding as the polymorphic identity function. All other rules simply recursively translate subterms.

We first prove that this translation preserves falseness. We encode False in CC-CC as $\Pi\, \mathrm{A} : \star.\ \mathrm{A}$. This encoding represents a function that takes any arbitrary proposition A and returns a proof of A. Similar, in CC False as $\Pi\, \mathrm{A} : \star.\ \mathrm{A}$.

It is clear from [M-Prod-*] that the translation preserves falseness. We use = as the terms are not just definitionally equivalent, but syntactically identical.

Lemma 2.1 (False Preservation). $\mathbf{False}^{\circ} = \mathsf{False}$

To prove type preservation, we split the proof into three key lemmas. First, we show *compositionality*, *i.e.*, that the translation from CC-CC to CC commutes with substitution. Then we prove preservation of reduction semantics and equivalence, which essentially follows from compositionality. Finally, we prove type preservation, which relies on preservation of equivalence and on compositionality.

Compositionality is an important lemma since the type system and conversion relations are defined by substitution.

Lemma 2.2 (Compositionality). $(\mathbf{e}[\mathbf{e'}/\mathbf{x}])^{\circ} = \mathbf{e}^{\circ}[\mathbf{e'}^{\circ}/\mathsf{x}]$

Proof. The proof is by induction on the typing derivation $\Gamma \vdash \mathbf{e} : \mathbf{A}$. Recall that by convention this derivation is an implicit argument to the lemma.

Case [Ax-*] Trivial, since $\mathbf{e} = \star$ cannot have free variables.

Case [Var] Hence $\mathbf{e} = \mathbf{x'}$. There are two subcases:

Sub-case $\mathbf{x'} = \mathbf{x}$ Then the proof follows since $(\mathbf{x}[\mathbf{e'}/\mathbf{x}])^{\circ} = \mathbf{e'}^{\circ} = \mathsf{x}[\mathbf{e'}^{\circ}/\mathsf{x}]$

Sub-case $\mathbf{x'} \neq \mathbf{x}$ Then the proof follows since $(\mathbf{x'}[\mathbf{e'}/\mathbf{x}])^{\circ} = \mathsf{x'} = \mathsf{x'}[\mathbf{e'}^{\circ}/\mathsf{x}]$

Case [Let] Follows easily by the inductive hypotheses, since both the translation of $\mathbf{let}$ and the definition of substitution are structural, except for the capture avoidance reasoning.

Case [Prod-*] Follows easily by the inductive hypotheses, since both the translation of $\mathbf{\Pi}$ and the definition of substitution are structural, except for the capture avoidance reasoning.

$\vdots$

Case [Conv] Recall that the translation is defined by induction on typing derivations, and therefore we have the conversion typing rule:

$$\dfrac{\Gamma \vdash \mathbf{e} : \mathbf{A} \qquad \Gamma \vdash \mathbf{B} : \mathbf{U} \qquad \Gamma \vdash \mathbf{A} \equiv \mathbf{B}}{\Gamma \vdash \mathbf{e} : \mathbf{B}}\ [\textsc{Conv}]$$

We must show that $(\mathbf{e}[\mathbf{e'}/\mathbf{x}])^{\circ} \equiv \mathbf{e}^{\circ}[\mathbf{e'}^{\circ}/\mathsf{x}]$ at, loosely speaking, the type $\mathbf{B}^{\circ}$. (Loosely, since we haven't show type preservation yet.)

By the induction hypothesis applied to $\Gamma \vdash \mathbf{e} : \mathbf{A}$, we have that $(\mathbf{e}[\mathbf{e'}/\mathbf{x}])^{\circ} \equiv \mathbf{e}^{\circ}[\mathbf{e'}^{\circ}/\mathsf{x}]$ at the type $\mathbf{A}^{\circ}$.

The astute type theorist may be concerned that we first need to show that these terms are well-typed—*i.e.*, that we need to show type preservation—and that $\mathbf{A}^{\circ} \equiv \mathbf{B}^{\circ}$, *i.e.*, *coherence*. However, our definition of equivalence in CC and CC-CC is based on the CIC *untyped* equivalence [5, Chapter 4], so the proof is already done. We can think of this equivalence as justifying semantic equivalences that are statically ruled out by a conservative syntactic type system. The advantage of this equivalence is that it allows us to stage the proof as we have.

□

Next we show that the translation preserves reduction, or that our model in CC weakly simulates reduction in CC-CC. This is used both to show that equivalence is preserved, since equivalence is defined by reduction, and to show type safety.

Lemma 2.3 (Pres. of Reduction). *If* $\mathbf{e} \triangleright \mathbf{e'}$ *then* $\mathbf{e}^{\circ} \triangleright^{*} \mathbf{e'}^{\circ}$

Proof. By cases on $\mathbf{e} \triangleright \mathbf{e'}$. The only interesting case is for the reduction of closures.

Case $\langle\langle(\boldsymbol{\lambda}\, \mathbf{x'} : \mathbf{A'}, \mathbf{x} : \mathbf{A}.\, \mathbf{e}_b), \mathbf{e'}\rangle\rangle\ \mathbf{e} \triangleright_{\beta} \mathbf{e}_b[\mathbf{e'}/\mathbf{x'}][\mathbf{e}/\mathbf{x}]$

We must show that

$(\langle\langle(\boldsymbol{\lambda}\, \mathbf{x'} : \mathbf{A'}, \mathbf{x} : \mathbf{A}.\, \mathbf{e}_b), \mathbf{e'}\rangle\rangle\ \mathbf{e})^{\circ} \triangleright^{*} (\mathbf{e}_b[\mathbf{e'}/\mathbf{x'}][\mathbf{e}/\mathbf{x}])^{\circ}$

$$\begin{aligned}
& (\langle\langle(\boldsymbol{\lambda}\, \mathbf{x'} : \mathbf{A'}, \mathbf{x} : \mathbf{A}.\, \mathbf{e}_b), \mathbf{e'}\rangle\rangle\ \mathbf{e})^{\circ} && \tag{1}\\
& = ((\lambda\, \mathsf{x'} : \mathbf{A'}^{\circ}.\, \lambda\, \mathsf{x} : \mathbf{A}^{\circ}.\, \mathbf{e}_b^{\circ})\ \mathbf{e'}^{\circ})\ \mathbf{e}^{\circ} && \text{by definition} \tag{2}\\
& \triangleright_{\beta}^{2}\ \mathbf{e}_b^{\circ}[\mathbf{e'}^{\circ}/\mathsf{x'}][\mathbf{e}^{\circ}/\mathsf{x}] && \tag{3}\\
& = (\mathbf{e}_b[\mathbf{e'}/\mathbf{x'}][\mathbf{e}/\mathbf{x}])^{\circ} && \text{by Lemma 2.2} \tag{4}
\end{aligned}$$

□

Now we show that reduction *sequences* are preserved. This essentially follows from preservation of single-step reduction, Lemma 2.3.

Lemma 2.4 (Preservation of Reduction Sequences). *If* $\mathbf{e} \rhd^* \mathbf{e}'$ *then* $\mathbf{e}^\circ \rhd^* \mathbf{e}'^\circ$

Proof. The proof is by induction on the length of the reduction sequence $\mathbf{e} \rhd^* \mathbf{e}'$. The base case is trivial, and the inductive case follows by Lemma 2.3 (Pres. of Reduction) and the inductive hypothesis. □

Next, we show *coherence*, *i.e.*, that the translation preserves equivalence. The proof essentially follows from Lemma 2.4, but we must show that our η rule for closures is preserved.

Lemma 2.5 (Coherence). *If* $\mathbf{e}_1 \equiv \mathbf{e}_2$ *then* $\mathbf{e}_1^\circ \equiv \mathbf{e}_2^\circ$

Proof. The proof is by induction on the derivation $\mathbf{e} \equiv \mathbf{e}'$. The only interesting case is for η equivalence of closures.

Case [≡] Follows by Lemma 2.4 (Preservation of Reduction Sequences).

Case [≡-Clo$_1$]

By assumption, we have the following.

(1) $\mathbf{e}_1 \rhd^* \langle\!\langle \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_1', \mathbf{e}' \rangle\!\rangle$

(2) $\mathbf{e}_2 \rhd^* \mathbf{e}_2'$

(3) $\mathbf{e}_1[\mathbf{e}'/\mathbf{x}'] \equiv \mathbf{e}_2'\ \mathbf{x}$

We must show that $\mathbf{e}_1^\circ \equiv \mathbf{e}_2^\circ$. By [≡-$\eta_1$], it suffices to show:

(1) $\mathbf{e}_1^\circ \rhd^* \lambda\, x : \mathbf{A}^\circ[\mathbf{e}'^\circ/x'].\, \mathbf{e}_1'^\circ[\mathbf{e}'^\circ/x']$, which follows since:

$$\begin{aligned}
\mathbf{e}_1^\circ\ &\rhd^* (\langle\!\langle \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_1', \mathbf{e}' \rangle\!\rangle)^\circ && \text{by Lemma 2.4} && (5)\\
&= (\lambda\, x' : \mathbf{A}'^\circ.\, \lambda\, x : \mathbf{A}^\circ.\, \mathbf{e}_1'^\circ)\ \mathbf{e}'^\circ && && (6)\\
&\rhd \lambda\, x : \mathbf{A}'^\circ[\mathbf{e}'^\circ/x'].\, \mathbf{e}_1'^\circ[\mathbf{e}'^\circ/x'] && && (7)
\end{aligned}$$

(2) $\mathbf{e}_2^\circ \rhd^* \mathbf{e}_2'^\circ$ which follows by Lemma 2.4.

(3) $\mathbf{e}_1'^\circ[\mathbf{e}'^\circ/x'] \equiv \mathbf{e}_2'^\circ\ x$, which follows by the inductive hypothesis applied to $\mathbf{e}_1[\mathbf{e}'/\mathbf{x}'] \equiv \mathbf{e}_2'\ \mathbf{x}$ and Lemma 2.2.

Case [≡-Clo$_2$] is symmetric.

□

We can now show our final lemma: type preservation.

Lemma 2.6 (Type Preservation).

(1) If $\vdash \boldsymbol{\Gamma}$ *then* $\vdash \boldsymbol{\Gamma}^\circ$

(2) If $\boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ *then* $\boldsymbol{\Gamma}^\circ \vdash \mathbf{e}^\circ : \mathbf{A}^\circ$

Proof. We prove parts 1 and 2 simultaneously by induction on the mutually defined judgments $\vdash \boldsymbol{\Gamma}$ and $\boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{A}$. Most cases follow easily by the induction hypothesis.

Case [W-Empty] Trivial.

Case [W-Def] We must show that $\vdash (\boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A})^\circ$. By [W-Def] in CC and part 1 of the inductive hypothesis, it suffices to show that $\boldsymbol{\Gamma}^\circ \vdash \mathbf{e}^\circ : \mathbf{A}^\circ$, which follows by part 2 of the inductive hypothesis applied to $\boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{A}$.

Case [W-Assum] We must show that $\vdash (\boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A})^\circ$. By [W-Assum] in CC and part 1 of the inductive hypothesis, it suffices to show that $\boldsymbol{\Gamma}^\circ \vdash \mathbf{A}^\circ : \mathbf{U}^\circ$, which follows by part 2 of the inductive hypothesis applied to $\boldsymbol{\Gamma} \vdash \mathbf{A} : \mathbf{U}$.

Case [Ax-*] It suffices to show that $\vdash \boldsymbol{\Gamma}^\circ$, since $\star^\circ = \star$, which follows by part 1 of the inductive hypothesis.

⋮

Case [T-Code-*]

We have that

$$\frac{\boldsymbol{\Gamma} \vdash \mathbf{A}' : \mathbf{U}' \qquad \boldsymbol{\Gamma}, \mathbf{x}' : \mathbf{A}' \vdash \mathbf{A} : \mathbf{U} \qquad \boldsymbol{\Gamma}, \mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A} \vdash \mathbf{B} : \star}{\boldsymbol{\Gamma} \vdash \mathbf{Code}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B} : \star}$$

We must show that $\boldsymbol{\Gamma}^\circ \vdash \Pi\, x' : \mathbf{A}'^\circ.\, \Pi\, x : \mathbf{A}^\circ.\, \mathbf{B}^\circ : \star$

By two applications of [Prod-*], it suffices to show

- $\boldsymbol{\Gamma}^\circ \vdash \mathbf{A}'^\circ : \mathbf{U}'^\circ$, which follows by part 2 of the inductive hypothesis.

- $\Gamma^{\circ}, x' : A'^{\circ} \vdash A^{\circ} : U^{\circ}$, which follows by part 2 of the inductive hypothesis.
- $\Gamma^{\circ}, x' : A'^{\circ}, x : A^{\circ} \vdash B^{\circ} : \star$, which follows by part 2 of the inductive hypothesis and by definition that $\star^{\circ} = \star$

Case [Code]

We have that

$$\frac{\Gamma, x' : A', x : A \vdash e : B}{\Gamma \vdash \lambda\, x' : A', x : A.\, e : \mathrm{Code}\,(x' : A', x : A).\, B}$$

By definition of the translation, we must show $\Gamma^{\circ} \vdash \lambda\, x' : A'^{\circ}.\, \lambda\, x : A^{\circ}.\, e^{\circ} : \Pi\, x' : A'^{\circ}.\, \Pi\, x : A^{\circ}.\, B^{\circ}$, which follows by two uses of [Lam] in CC and part 2 of the inductive hypothesis.

Case [Clo]

We have that

$$\frac{\Gamma \vdash e : \mathrm{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash \langle\langle e, e' \rangle\rangle : \Pi\, x : A[e'/x'].\, B[e'/x']}$$

By definition of the translation, we must show that $\Gamma^{\circ} \vdash e^{\circ}\ e'^{\circ} : (\Pi\, x : A[e'/x'].\, B[e'/x'])^{\circ}$.

By Lemma 2.2 (Compositionality), it suffices to show that $\Gamma^{\circ} \vdash e^{\circ}\ e'^{\circ} : \Pi\, x : A^{\circ}[e'^{\circ}/x'].\, B^{\circ}[e'^{\circ}/x']$.

By [App] in CC, it suffices to show that

- $\Gamma^{\circ} \vdash e^{\circ} : \Pi\, x' : A'.\, \Pi\, x : A^{\circ}.\, B^{\circ}$, which follows with $A' = A'^{\circ}$ by part 2 of the inductive hypothesis.
- $\Gamma^{\circ} \vdash e'^{\circ} : A'$, which follows by part 2 of the inductive hypothesis.

Case [App] Similar to the case for [Clo].

Case [Conv] Follows by part 2 of the inductive hypothesis and Lemma 2.5 (Coherence).

□

Finally, we can prove the desired consistency and type safety theorems.

Theorem 2.7 (Consistency of CC-CC). *There does not exist a closed expression* e *such that* $\cdot \vdash e : \mathrm{False}$.

Type safety tells us that there is no undefined behavior that causes a program to get stuck before it produces a value, and all programs terminate.

Theorem 2.8 (Type Safety of CC-CC). *If* $\cdot \vdash e : A$, *then* $e \triangleright^{*} v$ *and* $v \not\triangleright v'$.

$\boxed{\Gamma \vdash e : t \leadsto \mathbf{e} \text{ where } \Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash \star : \square \leadsto \boldsymbol{\star}}\ [\text{CC-*}] \qquad \frac{}{\Gamma \vdash x : A \leadsto \mathbf{x}}\ [\text{CC-Var}]$$

$$\frac{\Gamma \vdash e : A \leadsto \mathbf{e} \qquad \Gamma \vdash A : U \leadsto \mathbf{A} \qquad \Gamma, x : A \vdash e' : B \leadsto \mathbf{e'}}{\Gamma \vdash \mathrm{let}\, x = e : A \,\mathrm{in}\, e' : B[e/x] \leadsto \mathbf{let\, x = e : A\, in\, e'}}\ [\text{CC-Let}]$$

$$\frac{\Gamma \vdash A : U \leadsto \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \leadsto \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \star \leadsto \boldsymbol{\Pi\, x : A.\, B}}\ [\text{CC-Prod-*}] \qquad \frac{\Gamma \vdash A : U \leadsto \mathbf{A} \qquad \Gamma, x : A \vdash B : \square \leadsto \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \square \leadsto \boldsymbol{\Pi\, x : A.\, B}}\ [\text{CC-Prod-}\square]$$

$$\frac{\Gamma, x : A \vdash B : U \leadsto \mathbf{B} \qquad \begin{array}{c} \Gamma, x : A \vdash e : B \leadsto \mathbf{e} \qquad \Gamma \vdash A : U \leadsto \mathbf{A} \\ x_i : A_i \ldots = \mathrm{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma) \end{array} \qquad \Gamma \vdash A_i : U \leadsto \mathbf{A}_i \ldots}{\begin{array}{c} \Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B \leadsto \langle\!\langle(\boldsymbol{\lambda}\, (\mathbf{n} : \boldsymbol{\Sigma}\, (\mathbf{x}_i : \mathbf{A}_i \ldots), \mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n\, in\, A}). \\ \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n\, in\, e}), \\ \langle \mathbf{x}_i \ldots \rangle\, \mathbf{as}\, \boldsymbol{\Sigma}\, (\mathbf{x}_i : \mathbf{A}_i \ldots))\rangle\!\rangle \end{array}}\ [\text{CC-Lam}]$$

$$\frac{\Gamma \vdash e_1 : \Pi\, x : A.\, B \leadsto \mathbf{e}_1 \qquad \Gamma \vdash e_2 : A \leadsto \mathbf{e}_2}{\Gamma \vdash e_1\; e_2 : B[e_2/x] \leadsto \mathbf{e}_1\; \mathbf{e}_2}\ [\text{CC-App}] \qquad \frac{\Gamma \vdash A : \star \leadsto \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \leadsto \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : \star \leadsto \boldsymbol{\Sigma\, x : A.\, B}}\ [\text{CC-Sig-*}]$$

$$\frac{\Gamma \vdash A : \square \leadsto \mathbf{A} \qquad \Gamma, x : A \vdash B : \square \leadsto \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : \star \leadsto \boldsymbol{\Sigma\, x : A.\, B}}\ [\text{CC-Sig-}\square] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B \leadsto \mathbf{e}}{\Gamma \vdash \mathrm{fst}\, e : A \leadsto \mathbf{fst\, e}}\ [\text{CC-Fst}]$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B \leadsto \mathbf{e}}{\Gamma \vdash \mathrm{snd}\, e : B[\mathrm{fst}\, e/x] \leadsto \mathbf{snd\, e}}\ [\text{CC-Snd}] \qquad \frac{\Gamma \vdash e : A \leadsto \mathbf{e}}{\Gamma \vdash e : B \leadsto \mathbf{e}}\ [\text{CC-Conv}]$$

$\boxed{\vdash \Gamma \leadsto \boldsymbol{\Gamma} \text{ where } \vdash \Gamma}$

$$\frac{}{\vdash \cdot \leadsto \cdot}\ [\text{W-Empty}] \qquad \frac{\vdash \Gamma \leadsto \boldsymbol{\Gamma} \qquad \Gamma \vdash A : \_ \leadsto \mathbf{A}}{\vdash \Gamma, x : A \leadsto \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A}}\ [\text{W-Assum}]$$

$$\frac{\vdash \Gamma \leadsto \boldsymbol{\Gamma} \qquad \Gamma \vdash A : \_ \leadsto \mathbf{A} \qquad \Gamma \vdash e : A \leadsto \mathbf{e}}{\vdash \Gamma, x = e : A \leadsto \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}}\ [\text{W-Def}]$$

Fig. 9. Closure Conversion

$$\begin{array}{rcl} \mathrm{FV}(e, B, \Gamma) & \overset{\text{def}}{=} & \Gamma_0, \ldots, \Gamma_n, x_0 : A_0, \ldots, x_n : A_n \\ & \textit{where} & x_0, \ldots, x_n = \mathrm{fv}(e, B) \\ & & \Gamma \vdash x_0 : A_0, \ldots, \Gamma \vdash x_n : A_n \\ & & \Gamma_0 = \mathrm{FV}(A_0, \_, \Gamma) \\ & & \vdots \\ & & \Gamma_n = \mathrm{FV}(A_n, \_, \Gamma) \end{array}$$

Fig. 10. CC Dependent Free Variable Sequences

3 CLOSURE CONVERSION

We present the closure conversion translation in Figure 9. We define the following notation for the translation of expressions.

$$e^+ \overset{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash e : A \rightsquigarrow \mathbf{e}$$

The CC-CC expression e^+ refers to the translation of the well-typed CC term e, with typing derivation for e as an implicit parameter.

Every case of the translation except for functions is trivial, including application [CC-App], since application is still the elimination form for closures after closure conversion. In the nontrivial case [CC-Lam], we translate CC functions to CC-CC closures. The translation of a function $\lambda\, x : A.\, e$ produces a closure $\langle\!\langle \mathbf{e}_1, \mathbf{e}_2 \rangle\!\rangle$. We compute the free variables (and their type annotations) of the function $\lambda\, x : A.\, e$, $x_i : A_i \ldots$, using the metafunction $\mathrm{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma)$ defined shortly. The first component $\mathbf{e}_1$ is closed code. Ignoring the type annotation for a moment, the code $\boldsymbol{\lambda}\,(\mathbf{n}, \mathbf{x}).\ \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ e^+$ projects each of the $|i|$ free variables $\mathbf{x}_i \ldots$ from the environment $\mathbf{n}$ and binds them in the scope of the body e^+. But CC-CC is dependently typed, so we also bind the free variables from the environment in the type annotation for the argument $\mathbf{x}$, *i.e.*, producing the annotation $\mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ A^+$ instead of just $\mathbf{x} : A^+$. Next we produce the environment type $\Sigma\,(\mathbf{x}_i : A^+ \ldots)$, from the free source variables $x_i \ldots$ of types $A_i \ldots$. We create the environment $\mathbf{e}_2$ by creating the dependent n-tuple $\langle \mathbf{x}_i \ldots \rangle$; these free variables will be replaced by values at run time.

To compute the sequence of free variables and their types, we define the metafunction $\mathrm{FV}(e, B, \Gamma)$ in Figure 10. Just from the syntax of terms e, B, we can compute some sequence of free variables $x_0, \ldots, x_n = \mathrm{fv}(e, B)$. However, the types of these free variables $A_0, \ldots, A_n$ may contain *other* free variables, and their types may contain still others, and so on! We must, therefore, recursively compute the a sequence of free variables and their types with respect to an environment Γ. Note that because the type B of a term e may contain different free variables than the term, we must compute the sequence with respect to both a term and its type. However, in all recursive applications of this metafunction—*e.g.*, $\mathrm{FV}(A_0, \_, \Gamma)$—the type of A_0 must be a universe and cannot have any free variables.

3.1 Type Preservation

First we prove type preservation, using the same staging as in Section 2. After we show type preservation, we show correctness of separate compilation. In CC, the lemmas required for type preservation do most of the work to allow us to prove correctness of separate compilation, since type checking includes reduction and thus we prove preservation of reduction sequences.

We first show *compositionality.* This lemma, which establishes that translation commutes with substitution, is the key difficulty in our proof of type preservation because closure conversion internalizes free variables. Whether we substitute a term for a variable before or after translation can drastically affect the shape of closures produced by the translation. For instance, consider the term $(\lambda\, y : A.\, e)[e'/x]$. If we perform this substitution before translation, then we will generate an environment with the shape $\langle \mathbf{x}_i \ldots, \mathbf{x}_j \ldots \rangle$, *i.e.*, with only free variables and without $\mathbf{x}$ in the environment. However, if we translate the individual components and then perform the substitution, then the environment will have the shape $\langle \mathbf{x}_i \ldots, e'^+, \mathbf{x}_j \ldots \rangle$—that is, $\mathbf{x}$ would be free when we create the environment and substitution would replace it by e'^+. We use our η-principle for closures to show that closures that differ in this way are still equivalent.

Lemma 3.1 (Compositionality). $(e_1[e_2/x])^+ \equiv e_1^+[e_2^+/x]$

Proof. By induction on the typing derivation for e_1. We give the key cases.

Case [Ax-Var]

We know that e_1 is some free variable x', so either $x' = x$, hence $e_2^+ \equiv e_2^+$, or $x' \neq x$, hence $x'^+ \equiv x'^+$.

Case [T-Code-*]

We know that $e_1 = \Pi\, x'{:}A.\ B$. W.l.o.g., assume $x' \neq x$. We must show $(\Pi\, x'{:}A[e_2/x].\ B[e_2/x])^+ \equiv (\Pi\, x' : A.\ B)^+[e_2^+/x]$.

$$\begin{aligned}
& (\Pi\, x' : A[e_2/x].\ B[e_2/x])^+ && (8)\\
&= \mathbf{\Pi\, x'} : (A[e_2/x])^+.\ (B[e_2/x])^+ && (9)\\
&\quad \text{by definition of the translation} &&\\
&= \mathbf{\Pi\, x'} : (A^+[e_2^+/\mathbf{x}]).\ (B^+[e_2^+/\mathbf{x}]) && (10)\\
&\quad \text{by the inductive hypothesis for } A \text{ and } B &&\\
&= (\mathbf{\Pi\, x'} : A^+.\ B^+)[e_2^+/\mathbf{x}] && (11)\\
&\quad \text{by definition of substitution} &&\\
&= (\Pi\, x' : A.\ B)^+[e_2^+/\mathbf{x}] && (12)\\
&\quad \text{by definition of translation} &&
\end{aligned}$$

Case [Prod-□]. Similar to [Prod-*]

Case [Lam]

We know that $e_1 = \lambda\, y{:}A.\ e$. W.l.o.g., assume that $y \neq x$. We must show that $((\lambda\, y : A.\ e)[e_2/x])^+ \equiv (\lambda\, y : A.\ e)^+[e_2^+/x]$. Recall that by convention we have that $\Gamma \vdash \lambda\, y : A.\ e : \Pi\, y : A.\ B$.

$$\begin{aligned}
& ((\lambda\, y : A.\ e)[e_2/x])^+ && (13)\\
&= (\lambda\, y : (A[e_2/x]).\ e[e_2/x])^+ && (14)\\
&\quad \text{by substitution} &&\\
&= \langle\!\langle (\boldsymbol{\lambda}\, \mathbf{n} : \Sigma\, (\mathbf{x}_i : A_i^+ \ldots), \mathbf{y} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ (A[e_2/x])^+. && \\
&\qquad \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ (e[e_2/x])^+), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle && (15)\\
&\quad \text{by definition of the translation} &&
\end{aligned}$$

where $x_i : A_i \ldots\ = \mathrm{FV}(\lambda\, y : (A[e_2/x]).\ e[e_2/x], \Gamma)$. Note that x is not in the sequence $(x_i \ldots)$. On the other hand, we have

$$\begin{aligned}
\mathbf{f} &= (\lambda\, y : A.\ e)^+[e_2^+/x] && (16)\\
&= \langle\!\langle (\boldsymbol{\lambda}\, \mathbf{n} : \Sigma\, (\mathbf{x}_j : A_j^+ \ldots), \mathbf{y} : \mathbf{let}\ \langle \mathbf{x}_j \ldots \rangle = \mathbf{n}\ \mathbf{in}\ A^+. && \\
&\qquad \mathbf{let}\ \langle \mathbf{x}_j \ldots \rangle = \mathbf{n}\ \mathbf{in}\ e^+), \langle \mathbf{x}_{j_0} \ldots, e_2^+, \mathbf{x}_{j_{i+1}} \ldots \rangle \rangle\!\rangle && (17)\\
&\quad \text{by definition of the translation} &&
\end{aligned}$$

where $x_j : A_j \ldots\ = \mathrm{FV}(\lambda\, y : A.\ e, \Gamma)$. Note that x is in $x_j \ldots$; we can write the sequence as $(x_{j_0} \ldots x, x_{j_{i+1}} \ldots)$. Therefore, the environment we generate contains e_2^+ in position j_i.
By [≡-Clo$_1$], it suffices to show that
$\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ (e[e_2/x])^+ \equiv \mathbf{f\ y}$ where $\mathbf{f}$ is the closure from Equation (16).

$$\begin{aligned}
\mathbf{f\ y} &\equiv \mathbf{let}\ \langle \mathbf{x}_{j_0} \ldots \mathbf{x}, \mathbf{x}_{j_{i+1} \ldots} \rangle = \langle \mathbf{x}_{j_0} \ldots, e_2^+, \mathbf{x}_{j_{i+1}} \ldots \rangle\ \mathbf{in}\ e^+ && (18)\\
&\quad \text{by } \rhd_\beta \text{ in CC-CC} &&\\
&\equiv e^+[e_2^+/\mathbf{x}] && (19)\\
&\quad \text{by } |j| \text{ applications of } \rhd_\zeta \text{, since only } \mathbf{x} \text{ has a value} &&\\
&\equiv (e[e_2/x])^+ && (20)\\
&\quad \text{by the inductive hypothesis applied to the derivation for } e &&
\end{aligned}$$

$$\equiv \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ (e[e_2/x])^+ \tag{21}$$

by |i| applications of $\triangleright_\zeta$, since no variable has a value

□

Next we show that if a source term e takes a step, then its translation e^+ reduces in some number of steps to a definitionally equivalent term $\mathbf{e}$. This proof essentially follows by Lemma 3.1. Then we show by induction on the length of the reduction sequence that the translation preserves reduction sequences. Note that since Lemma 3.1 relies on our η equivalence rule for closures, we can only show reduction up to definitional equivalence. That is, we cannot show $e^+ \triangleright^* e'^+$. This is not a problem; we reason about source programs to equivalence anyway, and not up to syntactic equality.

Lemma 3.2 (Preservation of Reduction). *If* $\Gamma \vdash e \triangleright e'$ *then* $\Gamma^+ \vdash e^+ \triangleright^* \mathbf{e}$ *and* $\mathbf{e} \equiv e'^+$

Proof. By cases on $\Gamma \vdash e \triangleright e'$. Most cases follow easily by Lemma 3.1, since most cases of reduction are defined by substitution.

Case $x \triangleright_\delta e'$ where $x = e' : A \in \Gamma$.

We must show that $\mathbf{x} \triangleright^* \mathbf{e}$ and $e'^+ \equiv \mathbf{e}$. Let $\mathbf{e} \overset{\text{def}}{=} e'^+$. It suffices to show that $\mathbf{x} \triangleright^* e'^+$. By definition of the translation, we know that $\mathbf{x} = e'^+ : A^+ \in \Gamma^+$ and $\mathbf{x} \triangleright_\delta e'^+$.

Case $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleright_\zeta e_2[e_1/x]$

We must show that $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)^+ \triangleright^* \mathbf{e}$ and $(e_2[e_1/x])^+ \equiv \mathbf{e}$. Let $\mathbf{e} \overset{\text{def}}{=} e_2^+[e_1^+/\mathbf{x}]$.

$$(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)^+ = \mathbf{let}\ \mathbf{x} = e_1^+\ \mathbf{in}\ e_2^+ \quad \text{by definition of the translation} \tag{22}$$

$$\triangleright_\zeta e_2^+[e_1^+/\mathbf{x}] \tag{23}$$

$$\equiv (e_2[e_1/x])^+ \tag{24}$$

byLemma 3.1 (Compositionality)

Case $(\lambda\, x : A.\, e_1)\ e_2 \triangleright_\beta e_1[e_2/x]$

We must show that $((\lambda\, x : A.\, e_1)\ e_2)^+ \triangleright^* \mathbf{e}$ and $(e_2[e_1/x])^+ \equiv \mathbf{e}$. Let $\mathbf{e} \overset{\text{def}}{=} e_1^+[e_2^+/x]$.
By definition of the translation, $((\lambda\, x : A.\, e_1)\ e_2)^+ = \mathbf{f}\ e_2^+$, where

$$\mathbf{f} = \langle\!\langle (\boldsymbol{\lambda}\, \mathbf{n} : \Sigma\, (\mathbf{x}_i : A_i^+ \ldots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ A^+. \tag{25}$$

$$\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ e_1^+), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle \tag{26}$$

and where $x_i : A_i \ldots = \mathrm{FV}(\lambda\, x : A.\, e_1, \Gamma)$.
To complete the proof, observe that,

$$\mathbf{f}\ e_2^+ \triangleright_\beta \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ e_1^+[e_2^+/\mathbf{x}] \tag{27}$$

$$\triangleright_\zeta^{|i|} e_1^+[e_2^+/\mathbf{x}] \tag{28}$$

$$\equiv (e_1[e_2/x])^+ \quad \text{by Lemma 3.1} \tag{29}$$

□

Lemma 3.3 (Preservation of Reduction Sequences). *If* $\Gamma \vdash e \triangleright^* e'$ *then* $\Gamma^+ \vdash e^+ \triangleright^* \mathbf{e}$ *and* $\Gamma^+ \vdash \mathbf{e} \equiv e'^+$.

Proof. By induction on the length of the reduction sequence n.

Case $n = 0$ Therefore $e' = e$.
Let $\mathbf{e} = e^+$
By definition, $e^+ \triangleright^0 e^+$ and $e^+ \equiv e^+$ by reflexivity.

Case $n = i + 1$ By assumption, $\Gamma \vdash e \triangleright e_1$ and $\Gamma \vdash e_1 \triangleright^* e'$.
It suffices to show that $e^+ \triangleright^* \mathbf{e}_1$ and $\mathbf{e}_1 \equiv e'^+$.
By Lemma 3.2, $e^+ \triangleright^* \mathbf{e}_1$ and $\mathbf{e}_1 \equiv e_1^+$. Note that our notation for translation, $^+$, requires that we have a typing derivation for e_1, thus here we rely on subject reduction of CC to know that such a derivation exists.
It remains to be show that $\mathbf{e}_1 \equiv e'^+$.
By the induction hypothesis, $e_1^+ \triangleright^* \mathbf{e}$ and $\mathbf{e} \equiv e'^+$.

Since $e_1^+ \rhd^* \mathbf{e}$, by [$\equiv$], $e_1^+ \equiv \mathbf{e}$.
The goal follows by transitivity: $\mathbf{e}_1 \equiv e_1^+ \equiv \mathbf{e} \equiv e'^+$, therefore $e_1 \equiv e'^+$.

□

We can now show *coherence*, *i.e.*, that equivalent terms are translated to equivalent terms. As equivalence is defined primarily by $\rhd^*$, the only interesting part of the next proof is preserving η equivalence. To show that η equivalence is preserved, we require our new η rules for closures.

Lemma 3.4 (Coherence). *If* $\Gamma \vdash e \equiv e'$, *then* $\Gamma^+ \vdash e^+ \equiv e'^+$.

Proof. By induction on the $e \equiv e'$ judgment.

Case [$\equiv$]

By assumption, $e \rhd^* e_1$ and $e' \rhd^* e_1$.
By Lemma 3.3, $e^+ \rhd^* \mathbf{e}$ and $\mathbf{e} \equiv e_1^+$, and similarly. $e'^+ \rhd^* \mathbf{e}'$ and $\mathbf{e}' \equiv e_1^+$. The result follows by symmetry and transitivity.

Case [$\equiv$-η_1]

By assumption, $e \rhd^* \lambda\, x : t.\, e_1$, $e' \rhd^* e_2$ and $e_1 \equiv e_2\ x$.
Must show $e^+ \equiv e'^+$.
By Lemma 3.3, $e^+ \rhd^* \mathbf{e}$ and $\mathbf{e} \equiv (\lambda\, x : t.\, e_1)^+$, and similarly $e'^+ \rhd^* \mathbf{e}'$ and $\mathbf{e}' \equiv e_2^+$.
By transitivity of $\equiv$, it suffices to show $(\lambda\, x : t.\, e_1)^+ \equiv e_2^+$.
By definition of the translation,

$$(\lambda\, x : t.\, e_1)^+ = \langle\!\langle (\boldsymbol{\lambda}\, \mathbf{n} : \Sigma\, (\mathbf{x}_i : A_i^+ \ldots), \mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ A^+.\ \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ e_1^+), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle$$

where $x_i : A_i \ldots\ = \mathrm{FV}(\lambda\, x : t.\, e_1, \Gamma)$.
By [$\equiv$-Clo$_1$] in CC-CC, it suffices to show that

$$\begin{aligned} & \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ e_1^+ && (30)\\ \equiv\ & e_1^+ && (31)\\ & \text{by } |\mathrm{i}| \text{ applications of } \rhd_\zeta \\ \equiv\ & e_2^+\ \mathbf{x} && (32)\\ & \text{by the inductive hypothesis applied to } e_1 \equiv e_2\ x \end{aligned}$$

Case [$\equiv$-η_2] Symmetric to the previous case; requires [$\equiv$-η_2] instead of [$\equiv$-η_1].

□

Now we can prove type preservation. We give the technical version of the lemma required to complete the proof, followed by the desired statement of the theorem.

Lemma 3.5 (Type Preservation (technical)).

(1) If $\vdash \Gamma$ *then* $\vdash \Gamma^+$
(2) If $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash e^+ : A^+$

Proof. Parts 1 and 2 proven simultaneously by induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash e : A$. Part 1 follows easily by induction and part 2. We give the key cases for part 2.

Case [Lam]

We have that $\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B$. We must show that $\Gamma^+ \vdash (\lambda\, x : A.\, e)^+ : (\Pi\, x : A.\, B)^+$.
By definition of the translation, we must show that
$\langle\!\langle (\boldsymbol{\lambda}\, (\mathbf{n} : \Sigma\, (\mathbf{x}_i : A_i^+ \ldots), \mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ A^+).\ : \boldsymbol{\Pi}\, \mathbf{x} : A^+.\, B^+$
$\mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ e_1^+), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle$
where $x_i : A_i \ldots\ = \mathrm{FV}(\lambda\, x : t.\, e_1, \Gamma)$.
Notice that the annotation in the term $\mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ A^+$, does not match the annotation in the type $\mathbf{x} : A^+$. However, by [Clo], we can derive that the closure has type:
$\boldsymbol{\Pi}\, (\mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ A^+).\ (\mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ B^+)$,

This is equivalent to $\mathbf{\Pi\, x : A^+ .\, B^+}$ (under Γ^+), since $(\mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \langle \mathbf{x}_i \dots \rangle\ \mathbf{in}\ A^+) \equiv A^+$ as we saw in earlier proofs. So, by [Clo] and [Conv], it suffices to show that the environment and the code are well-typed.

By part 1 of the induction hypothesis applied (since each of $x_i : A_i \dots$ come from Γ), we know the environment is well-typed: $\Gamma^+ \vdash \langle \mathbf{x}_i \dots \rangle : \Sigma\, (\mathbf{x}_i : A_i^+ \dots)$.

Now we show that the code

$(\boldsymbol{\lambda}\, (\mathbf{n} : \Sigma\, (\mathbf{x}_i : A_i^+ \dots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ A^+).$

$\quad \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ e_1^+)$

has type $\mathbf{Code}\, (\mathbf{n}, \mathbf{x}).\ \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ B^+$. For brevity, we omit the duplicate type annotations on $\mathbf{n}$ and $\mathbf{x}$.

Observe that by the induction hypothesis applied to $\Gamma \vdash A : U$ and by weakening

$\mathbf{n} : \Sigma\, (\mathbf{x}_i : A_i^+ \dots) \vdash \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ A^+ : U^+$.

Hence, by [Code], it suffices to show

$\cdot, \mathbf{n}, \mathbf{x} \vdash \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ e_1^+ : \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ B^+$

which follows by the inductive hypothesis applied to $\Gamma, x : A \vdash e_1 : B$, and by weakening, since $x_i \dots$ are the free variables of e_1, A, and B.

Case [App]

We have that $\Gamma \vdash e_1\ e_2 : B[e_2/x]$. We must show that $\Gamma^+ \vdash e_1^+\ e_2^+ : (B[e_2/x])^+$. By Lemma 3.1, it suffices to show $\Gamma^+ \vdash e_1^+\ e_2^+ : B^+[e_2^+/\mathbf{x}]$, which follows by [App] and the inductive hypothesis applied to e_1, e_2 and B. □

Theorem 3.6 (Type Preservation). *If* $\Gamma \vdash e : t$ *then* $\Gamma^+ \vdash e^+ : t^+$.

3.2 Correctness

We prove *correctness of separate compilation* and *whole program correctness*. These two theorems follow easily from Lemma 3.3, but requires a little more work to state formally.

First, we need an independent specification that relates source values to target values in CC-CC. We do this by adding ground types, such as Bool, to both languages and consider results related when they are the same boolean: $\mathrm{true} \approx \mathbf{true}$ and $\mathrm{false} \approx \mathbf{false}$. It is well known how specify more sophisticated notions of observations.

Next, we define components and linking. Components in both CC and CC-CC are well-typed open terms, *i.e.*, $\Gamma \vdash e : A$. We implement linking by substitution, and define valid closing substitutions γ as follows.

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

We extend the compiler to closing substitutions γ^+ by point-wise application of the translation.

Our separate compilation guarantee is that the translation of the source component e linked with substitution γ is equivalent to first compiling e and then linking with some $\boldsymbol{\gamma}$ that is definitionally equivalent to γ^+.

Theorem 3.7 (Correctness of Separate Compilation). *If* $\Gamma \vdash e : A$ *and* A *is a ground type,* $\Gamma \vdash \gamma$, $\Gamma^+ \vdash \boldsymbol{\gamma}$, $\gamma(e) \triangleright^* v$, *and* $\gamma^+ \equiv \boldsymbol{\gamma}$ *then* $\boldsymbol{\gamma}(e^+) \triangleright^* \mathbf{v'}$ *and* $v^+ \approx \mathbf{v'}$

Proof. Since the translation commutes with substitution, preserves equivalence, reduction implies equivalence, and equivalence is transitive, the following diagram commutes.

$$\begin{tikzcd}
(\gamma(e))^+ \arrow[r, "\equiv"] \arrow[d, "\equiv"] & \boldsymbol{\gamma}(e^+) \arrow[d, "\equiv"] \\
v^+ \arrow[r, "\equiv"] & \mathbf{v'}
\end{tikzcd}$$

Since $\equiv$ on ground types implies $\approx$, we know that $v \approx \mathbf{v'}$. □

As a simple corollary, our compiler must also be whole-program correct. If a whole-program e evaluates to a value v, then the translation e^+ runs to a value equivalent to v^+.

Corollary 3.8 (Whole-Program Correctness). *If* $\cdot \vdash e : A$ *and* A *is a ground type, and* $e \triangleright^* v$ *then* $e^+ \triangleright^* \mathbf{v}$ *and* $v^+ \approx \mathbf{v}$

REFERENCES

[1] S. Boulier, P.-M. Pédrot, and N. Tabareau. The next 700 syntactical models of type theory. In *Conference on Certified Programs and Proofs (CPP)*, Jan. 2017. doi: 10.1145/3018610.3018620. URL https://hal.inria.fr/hal-01445835.

[2] T. Coquand. An analysis of Girard's paradox. In *Symposium on Logic in Computer Science (LICS)*, 1986. URL https://hal.inria.fr/inria-00076023.

[3] J. G. Hook and D. J. Howe. Impredicative strong existential equivalent to Type:Type. Technical report, Cornell University, 1986. URL http://hdl.handle.net/1813/6600.

[4] Z. Luo. ECC, an extended calculus of constructions. In *Symposium on Logic in Computer Science (LICS)*, 1989. doi: 10.1109/lics.1989.39193.

[5] The Coq Development Team. The Coq proof assistant reference manual, Oct. 2017. URL https://web.archive.org/web/20170109225110/https://coq.inria.fr/doc/Reference-Manual006.html.