JFP **32**, e12, 53 pages, 2022. © The Author(s), 2022. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

doi:10.1017/S0956796822000090

ANF preserves dependent types up to extensional equality

PAULETTE KORONKEVICH

University of British Columbia, Vancouver, British Columbia, Canada
(*e-mail:* pletrec@cs.ubc.ca)

RAMON RAKOW

University of British Columbia, Vancouver, British Columbia, Canada
(*e-mail:* ramon.rakow@alumni.ubc.ca)

AMAL AHMED

Northeastern University, Boston, MA 02115, USA
(*e-mail:* amal@ccs.neu.edu)

WILLIAM J. BOWMAN

University of British Columbia, Vancouver, British Columbia, Canada
(*e-mail:* wjb@williamjbowman.com)

Abstract

Many programmers use dependently typed languages such as Coq to machine-verify high-assurance software. However, existing compilers for these languages provide no guarantees after compiling, nor when linking after compilation. *Type-preserving compilers* preserve guarantees encoded in types and then use type checking to verify compiled code and ensure safe linking with external code. Unfortunately, standard compiler passes do not preserve the dependent typing of commonly used (intensional) type theories. This is because assumptions valid in simpler type systems no longer hold, and intensional dependent type systems are highly sensitive to syntactic changes, including compilation. We develop an A-normal form (ANF) translation with join-point optimization—a standard translation for making control flow explicit in functional languages—from the Extended Calculus of Constructions (ECC) with dependent elimination of booleans and natural numbers (a representative subset of Coq). Our dependently typed target language has equality reflection, allowing the type system to encode semantic equality of terms. This is key to proving type preservation and correctness of separate compilation for this translation. This is the first ANF translation for dependent types. Unlike related translations, it supports the universe hierarchy, and does not rely on parametricity or impredicativity.

1 Introduction

Dependently typed languages such as Coq, Agda, Idris, and F* allow programmers to write full-functional specifications for a program (or program component), implement the program, and prove that the program meets its specification. These languages have been widely used to build formally verified high-assurance software including the CompCert C

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

compiler (Leroy, 2009), the CertiKOS operating system kernel (Gu *et al*., 2015, 2016), and cryptographic primitives (Appel, 2015) and protocols (Barthe *et al*., 2009).

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

A verified compiler prevents miscompilation errors, since it is proven to preserve the run-time behavior of a program, but it cannot prevent linking errors. Note that linking errors can occur *even if S is compiled with a verified compiler*, since the external code we link with, C, is outside of the control of either the source language or the verified compiler. Ongoing work on CertiCoq (Anand *et al*., 2017) seeks to develop a *verified* compiler for Coq, but it cannot rule out linking with unsafe target code. One can develop simple examples in Coq that, once compiled to OCaml and linked with an unverified OCaml component, *jump to an arbitrary location in memory*—despite the Coq component being proven memory safe. We provide an example of this in the supplementary materials.

To rule out both miscompilation and linking errors, we could combine compiler verification with *type-preserving compilation*. A type-preserving compiler preserves types, representing specifications, into a target typed intermediate language (IL). The IL uses type checking at link time to enforce specifications when linking with external code, essentially implementing proof-carrying code (Necula, 1997). After linking in the IL, we have a whole program, so we can erase types and use verified compilation to machine code. To support safe linking with untyped code, we could use gradual typing to enforce safety at the boundary between the typed IL and untyped components (Ahmed, 2015; Lennon-Bertrand et al., 2022). Applied to Coq, this technique would provide significant confidence that the executable program P is as correct as the verified program S.

We consider this particular application of type preservation important for compiler verification and for protecting the trusted computing base when working in a dependently typed language, but type preservation has other long studied applications. Type-preserving compilers can provide a lightweight verification procedure using type-directed proof search (Chlipala, 2007), can be better equipped to handle optimizations and changes than a certified compiler (Tarditi *et al*., 1996; Chen *et al*., 2008), and can support debugging internal compiler passes (Peyton Jones, 1996).

Unfortunately, dependent-type preservation is difficult because compilation that preserves the *semantics* of the program may disrupt the *syntax-directed typing* of the program. This is particularly problematic with dependent types since expressions from a program end up in specifications expressed in the types. As the compiler transforms the syntax of the program, the expressions in types can become "out of sync" with the expressions in the compiled program.

To preserve dependent typing, the type system of the IL needs access to the semantic equivalences relied upon by compilation. Past work has used strong axioms, including

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

parametricity and impredicativity, to recover these equivalences (Bowman *et al.*, 2018). However, this approach does not support all dependent type system features, particularly predicative universe hierarchies (which are anti-impredicative) or ad hoc polymorphism and type quotation (which is anti-parametric) (Boulier *et al.*, 2017). Dependently typed languages, such as Coq, restrict impredicative quantification as it is inconsistent with other features and axioms, such as excluded middle and large elimination. Restricting impredicativity restricts type-level abstraction, so such languages often add a predicative universe hierarchy—abstractions over types live in a higher universe than the parameter, and so on. Higher universes enable recovering the ability to abstract over types and the types of types, etc., which is useful for generic programming and abstract mathematics. This means reliance on parametricity and impredicativity restrict which source programs can be compiled and thus cannot be applied in practice. To scale to a realistic dependently typed language such as Coq, we must encode these semantic equivalences in the type system without restricting core features.

To preserve dependent types, the compiler IL must encode the semantic equivalences relied upon by compilation. In this work, we do this using *extensionality*, which allows the type system to consider two types (or two terms embedded in a type) definitionally equivalent if the program contains an expression representing a proof that the two types (or terms) are equal. Using this feature, the translation can insert hints for the type system about which terms it can assume to be equivalent and provide proofs of those facts elsewhere to discharge these assumptions. This approach scales to higher universes and other features that prior work could not handle and does so without relying on parametricity or impredicativity in the target IL. Relying on extensionality has one key downside: decidable type checking becomes more complex. We discuss how to mitigate this downside in Section 7.

We present a dependent-type-preserving translation to A-normal form (ANF), a compiler intermediate representation that makes control flow explicit and facilitates optimizations (Sabry & Felleisen, 1992; Flanagan *et al.*, 1993). The source of this translation is ECC, the Extended Calculus of Constructions with dependent elimination of booleans and natural numbers. ECC represents a significant subset of Coq. The translation supports all core features of dependency, including higher universes, without relying on parametricity or impredicativity, in contrast to prior work (Bowman *et al.*, 2018; Cong & Asai, 2018*a*). This ensures that the translation works for existing dependently typed languages and that we can reuse existing work on ANF translations, such as join-point optimization. This work provides substantial evidence that dependent-type preservation can, in theory, scale to the practical dependently typed languages currently used for high-assurance software.

Our translation targets our typed IL CC_e^A, the ANF-restricted extensional Calculus of Constructions. CC_e^A features a machine-like semantics for evaluating ANF terms, and we prove correctness of separate compilation with respect to this machine semantics. To support the type-preserving translation of dependent elimination for booleans, CC_e^A uses two extensions to ECC: it records propositional equalities in typing derivations and applies equality reflection to access these equalities. These extensions make type checking undecidable. We discuss how to recover decidability in Section 7.

Our ANF translation is useful as a compiler pass, but it also provides insights into dependent-type preservation and ANF translations. The target IL is designed to express

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

and check semantic equivalences that the translation relies on for correctness. This lesson likely extends to other translations in a type-preserving compiler for dependent types.

We also develop a new proof architecture for reasoning about the ANF translation. This proof architecture is useful when ANF is achieved through a translation rather than a reduction system. Our translation is indexed by a *continuation*, a program with a hole, to build up the translated term. Thus, the type of the translated term can only be determined when the hole is filled with a well-typed term. Mirroring the translation, we use the *type* of the continuation to build up a proof of type-preservation.

This paper includes key definitions and proof cases; extended figures and proofs are available in Appendix 1.

2 Main ideas

ANF 101. A-normal form (ANF)[1] is a syntactic form that makes control flow explicit in the syntax of a program (Sabry & Felleisen, 1992; Flanagan *et al.*, 1993). ANF encodes *computation* (e.g., reducing an expression to a value) as a sequence of primitive intermediate computations composed through intermediate variables, similar to how all computation works in an assembly language.

For example, to reduce $\mathsf{snd}\ \mathsf{e}$, which projects the second element of a pair, we need to describe the evaluation order and control flow of each language primitive. We evaluate e to a value, and then, we project out the second component. ANF makes control flow explicit in the syntax by decomposing $\mathsf{snd}\ \mathsf{e}$ into the series of primitive computations that the machine must execute, sequenced by **let**. Roughly, we can think of the ANF translation $[\![\mathsf{snd}\ \mathsf{e}]\!]$ as $\mathbf{let}\ \mathbf{x} = [\![\mathsf{e}]\!]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}$. However, e could also be a deeply nested expression in the source language. In general, the ANF translation reassociates all the intermediate computations from $[\![\mathsf{e}]\!]$ so there are no nested **let** expressions, and we end up with $\mathbf{let}\ \mathbf{x}_1 = \mathbf{N}_1 \ldots \mathbf{x}_n = \mathbf{N}_n\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_n$.

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

Why Translation Disrupts Typing and How to Fix it. The problem with dependent-type preservation has little to do with ANF itself and everything to do with dependent types. Transformations which *ought* to be fine are not because the type theory is so beholden to details of syntax. This is essentially the problem of *commutative cuts* in type theory (Herbelin, 2009; Boutillier, 2012). Transformations that change the structure (syntax)

[1] The *A* in *A-normal form* has no further meaning. The name originates in a study of the (informal notion of) administrative redexes in CPS and formalizes these as a set *A* of source-to-source rewrites. The name *A-normal form* refers to the normal form with respect to the *A* reductions; the form is *A* normal. We emphasize this point because it is useful to think of ANF as syntactic form normal with respect to particular set of rewrites and not necessarily the output of a particular translation; we revisit this perspective in Section 7.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

of a program can disrupt *dependencies*. By dependency, we mean an expression e' whose type and evaluation depends on a sub-expression e. We call a sub-expression such as e *depended upon*. These dependencies occur in dependent elimination forms, such as application, projection, and if expressions. Transforming a dependent elimination can disrupt dependencies.

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

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

Notice that the *depended upon* sub-expression e is copied into the type, indicated by the solid line arrow. Dependent pairs can be used to define refinement types, such as encoding a type that guarantees an index is in bounds: $\Sigma\, \mathsf{x} : \mathsf{Int}.\, 0 < \mathsf{x} < \mathsf{len}\ \mathsf{e}'$. Then, the second projection $\mathsf{snd}\ \mathsf{e} : 0 < \mathsf{fst}\ \mathsf{e} < \mathsf{len}\ \mathsf{e}'$ represents an explicit proof about first projection. Unfortunately, transforming the expression $\mathsf{snd}\ \mathsf{e}$ can easily change its type.

For example, suppose we have the nested expression $\mathsf{f}\ (\mathsf{snd}\ \mathsf{e}) : \mathsf{C}$, and we want to let-bind all intermediate computations (which, incidentally, ANF does).

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

This is not well typed, even with the following dependent-let rule (treated as syntactic sugar for dependent application):

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

The problem now is the dependent elimination $\mathsf{snd}\ \mathsf{y}$ is let-bound and then used, changing the type to $\mathsf{B}[\mathsf{x} := \mathsf{fst}\ \mathsf{y}]$. This means the equality $\mathsf{y} = \mathsf{e}$ is missing,[2] but is needed to type check this term (indicated by the dotted line arrow). This fails since f expects $\mathsf{z} : \mathsf{B}[\mathsf{x} := \mathsf{fst}\ \mathsf{e}]$, but is applied to $\mathsf{z} : \mathsf{B}[\mathsf{x} := \mathsf{fst}\ \mathsf{y}]$. The typing rule essentially forgets that, by the time $\mathsf{snd}\ \mathsf{y}$ happens, the machine will have performed the step of computation $\mathsf{let}\ \mathsf{y} = \mathsf{e}$, forcing y to take on the value of e, so it *ought* to be safe to assume that $\mathsf{y} = \mathsf{e}$ in the types. When type checking in linearized machine languages, we need to record these machine steps throughout the typing derivation.

The above explanation applies to all dependent eliminations of negative types, such as dependently typed functions and compound data structures (modeled as Π and Σ types).

An analogous problem occurs with dependent elimination of positive types, which include data types eliminated through branching such as booleans with if expressions. In the dependent typing rule for if, the types of the branches learn whether the predicate of the if is either true or false. This allows the type system to statically reflect information from dynamic control flow.

$$\frac{\vdash \mathsf{e}_1 : \mathsf{B}[\mathsf{x} := \mathsf{true}] \qquad \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{x} := \mathsf{false}]}{\vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}[\mathsf{x} := \mathsf{e}]}$$

Now consider an if expression and a function f. The expression $\mathsf{f}\ (\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)$ is well typed, but if we want to push the application into the branches to make if behave a little more like `goto` (like the ANF translation does), this result is not well typed.

[2] This missing equality is added to most modern proof assistants, as explained further in the section.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$$\begin{array}{ll} \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ (\mathsf{f}\ \mathsf{e}_1) & \text{where } \mathsf{f} : (\mathsf{B}[\mathsf{x} := \mathsf{e}]) \to \mathsf{C} \\ \quad \mathsf{else}\ (\mathsf{f}\ \mathsf{e}_2) : \mathsf{C}[\mathsf{x} := \mathsf{e}] & \mathsf{e}_1 : \mathsf{B}[\mathsf{x} := \mathsf{true}] \\ & \mathsf{e}_2 : \mathsf{B}[\mathsf{x} := \mathsf{false}] \end{array}$$

This transformation fails when type checking the applications of f to the branches e_1 and e_2. The function f is expecting an argument of type $\mathsf{B}[\mathsf{x} := \mathsf{e}]$ but is applied to arguments of type $\mathsf{B}[\mathsf{x} := \mathsf{true}]$ and $\mathsf{B}[\mathsf{x} := \mathsf{false}]$. The type system cannot prove that e is equal to both true and false. In essence, this transformation relies on the fact that, by the time the expression $\mathsf{f}\ \mathsf{e}_1$ executes, the machine has evaluated e to true (and, analogously, e to false in the other branch), but the type system has no way to express this.

We design a **target language** type system in which we can express these intuitions and recover type preservation of these kinds of transformations. We use two extensions to ECC: one for negative types (Π and Σ in ECC) and one for positive types (booleans and natural numbers in ECC).

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

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

The highlighted part, $\mathbf{x} \overset{\delta}{=} \mathbf{e} : \mathbf{A}$, is the only difference from the standard dependent typing rule. This definition is introduced when type checking the body of the **let** and can be used to solve type equivalence in sub-derivations, instead of only in the substitution $\mathbf{A}'[\mathbf{x} := \mathbf{e}]$ in the "output" of the typing rule. While this is an extension to the type theory, it is a standard extension that is admissible in any Pure Type System (PTS) (Severi & Poll, 1994) and is a feature already found in dependently typed languages such as Coq. With this addition, the transformation of $(\mathsf{f}\ (\mathsf{snd}\ \mathsf{e}))$ type checks in the target language by recording the definition $\mathbf{y} \overset{\delta}{=} \mathbf{e} : \mathbf{A}$ while type checking the body of a **let** expression.

For positive types, we record a propositional equality between the term being eliminated and its value. For booleans, we need the following typing rule for **if**.[3]

$$\frac{\begin{array}{c}\Gamma, \mathbf{x} : \mathbf{Bool} \vdash \mathbf{B} : \mathbf{Type}_i \\ \Gamma \vdash \mathbf{e} : \mathbf{Bool} \qquad \Gamma, \boxed{\mathbf{p} : \mathbf{e} \equiv \mathbf{true}} \vdash \mathbf{e}_1 : \mathbf{B}[\mathbf{x} := \mathbf{true}] \qquad \Gamma, \boxed{\mathbf{p} : \mathbf{e} \equiv \mathbf{false}} \vdash \mathbf{e}_2 : \mathbf{B}[\mathbf{x} := \mathbf{false}]\end{array}}{\Gamma \vdash \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2 : \mathbf{B}[\mathbf{x} := \mathbf{e}]}$$

The two highlighted portions of the rule are modifications over the standard typing rule. This rule introduces a propositional equality $\mathbf{p}$ between the term $\mathbf{e}$ that appears in the calling context's type to the value known in the branches. This represents an assumption that $\mathbf{e}$ and **true** (or **false**) are equal in the type system and allows pushing the context surrounding the **if** expression into the branches.

Like definitions, propositional equalities thread "machine steps" into the typing derivations of $\mathbf{e}_1$ and $\mathbf{e}_2$. In contrast, these equalities are accessed through an additional type

[3] This rule essentially implements the convoy pattern (Chlipala, 2013). This pattern is common in dependent types; for example, transformation into the convoy pattern is used by Sozeau (2008) to implement the `Program` vernac in Coq to simplify reasoning with equality in function definitions.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

equivalence rule [≡-Reflect], which states that an equivalence holds between two terms if some proof of equality exists between them.

$$\frac{\Gamma \vdash \mathbf{p} : \mathbf{e}_1 \equiv \mathbf{e}_2}{\Gamma \vdash \mathbf{e}_1 \equiv \mathbf{e}_2} \text{ [}\equiv\text{-Reflect]}$$

Our earlier example if expression now type checks using the modified typing rule for $\mathbf{if}$ and [≡-Reflect]. We thread the equivalence $\mathbf{e} \equiv \mathbf{true}$ (respectively $\mathbf{e} \equiv \mathbf{false}$) which allows $\mathbf{f}$ to be applied to $\mathbf{e}_1$ (respectively $\mathbf{e}_2$) at the correct type.

Equality reflection is not a perfect solution, as adding this type equivalence rule can introduce undecidable type checking. While equality reflection allows definitions like $\mathbf{x} \overset{\delta}{=} \mathbf{e} : \mathbf{A}$ to be subsumed by a propositional equality $\mathbf{x} \equiv \mathbf{e}$, we use definitions when sufficient (such as for negative types) to avoid undecidable type checking. We discuss how to recover decidability of CC_e^A in Section 7; however, this would distract us from the ANF translation in the present work.

Formalizing Type-Preserving ANF Translation. Despite these simple extensions to CC_e^A, formalizing the ANF type-preservation argument is still tricky. In the source, looking at an expression such as $\mathsf{snd}\ \mathsf{e}$, we do not know whether the expression is embedded in a larger context. To formalize the ANF translation, it helps to have a compositional syntax for translating and reasoning about the types of an expression and the unknown context.

To make the translation compositional, we index the ANF translation by a target language (non-first-class) *continuation* $\mathbf{K}$ representing the rest of the computation in which a translated expression will be used.[4] A continuation $\mathbf{K}$ is a program with a hole (single linear variable) $[\cdot]$ and can be composed with a computation $\mathbf{N}$, written $\mathbf{K}[\mathbf{N}]$, to form a program $\mathbf{M}$. Keeping continuations non-first-class ensures that continuations must be used linearly and avoids control effects, which cause inconsistency with dependent types (Barthe & Uustalu, 2002; Herbelin, 2005). In ANF, there are only two continuations: either $[\cdot]$ or $\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M}$. Using continuations, we define ANF translation for Σ types and second projections as follows. We use $[\![\mathsf{e}]\!]$ as shorthand for translating e with an empty continuation, $[\![\mathsf{e}]\!]\ [\cdot]$.

$$[\![\Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}]\!]\ \mathbf{K} = \mathbf{K}[\Sigma\, \mathbf{x} : [\![\mathsf{A}]\!].\, [\![\mathsf{B}]\!]]$$
$$[\![\mathsf{snd}\ \mathsf{e}]\!]\ \mathbf{K} = [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{y}])$$

This allows us to focus on composing the primitive operations instead of reassociating $\mathbf{let}$ bindings.

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

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

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

[4] This is how ANF translation is implemented in Scheme by Flanagan *et al.* (1993), although their formal model is as a reduction system. An alternative implementation technique avoids continuations and instead returns a list of new declarations, similar to the allocation pass of Morrisett *et al.* (1999).

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Universes	U	$::=$	$\mathsf{Prop} \mid \mathsf{Type}_i$
Expressions	$\mathsf{e}, \mathsf{A}, \mathsf{B}$	$::=$	$\mathsf{x} \mid \mathsf{U} \mid \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \mid \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e} \mid \mathsf{e}\ \mathsf{e}$
		$\mid$	$\Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle\ \mathsf{as}\ \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \mid \mathsf{fst}\ \mathsf{e}$
		$\mid$	$\mathsf{snd}\ \mathsf{e} \mid \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{Bool}$
		$\mid$	$\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$
		$\mid$	$\mathsf{Nat} \mid \mathsf{zero} \mid \mathsf{succ}\ \mathsf{e} \mid \mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2$
Environments	Γ	$::=$	$\cdot \mid \Gamma, \mathsf{x} : \mathsf{A} \mid \Gamma, \mathsf{x} \stackrel{\delta}{=} \mathsf{e} : \mathsf{A}$

Fig. 1: ECC syntax.

is used.[5] We discuss this rule further in Section 4, and how continuation typing is not an extension to the target type theory.

The key lemma to prove type preservation is the following.

Lemma 2.1. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathbf{K} : ([\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]) \Rightarrow \mathbf{B}$*, then* $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![\mathsf{e}]\!]\ \mathbf{K} : \mathbf{B}$.

The intuition behind this lemma shows that type preservation follows if every time we construct a continuation $\mathbf{K}$, we show that $\mathbf{K}$ is well typed. The translation starts with an empty continuation $\mathbf{K}$, which is trivially well typed. We systematically build up the translation of e in $\mathbf{K}$ by recurring on sub-expressions of e and building up a new continuation $\mathbf{K}$. If each time we can show that this $\mathbf{K}$ is well typed, then by induction, the whole translation is type preserving. This lemma is indexed by an additional environment $\mathbf{\Gamma'}$ and answer type $\mathbf{B}$. Intuitively, this is because it is the continuation $\mathbf{K}$, not the expression e, that dictates the final answer type $\mathbf{B}$ of the translation, and the environment will provide some additional equalities and definitions in $\mathbf{\Gamma'}$ under which we can type check the transformed e.

3 Source: ECC with definitions

Our source language, ECC, is Luo's Extended Calculus of Constructions (ECC) (Luo, 1990) extended with dependent elimination of booleans, natural numbers with the recursive eliminator, and definitions (Severi & Poll, 1994). This language is a subset of CIC and is based on the presentation given in the Coq reference manual[6]; Timany & Sozeau (2017) give a recent account of the metatheory for the CIC, including all the features of ECC. 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 & Huet, 1988) with Σ types (strong dependent pairs) and an infinite predicative hierarchy of universes. There is no explicit phase distinction; that is, 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

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

[6] The Coq reference manual, https://coq.inria.fr/distrib/current/refman/language/cic.html, accessed 2021-07-07.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

types $\Sigma\, x : A.\, B$, dependent pairs $\langle e_1, e_2 \rangle$ as $\Sigma\, x : A.\, B$, first $\mathsf{fst}\ e$ and second $\mathsf{snd}\ e$ projections of dependent pairs, dependent let $\mathsf{let}\ x = e\ \mathsf{in}\ e'$, the boolean type Bool, the boolean values true and false, dependent if $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, the natural number type Nat, the natural number values zero and $\mathsf{succ}\ e$, and the recursive eliminator for natural numbers $\mathsf{indnat}\ A\ e\ e_1\ e_2$. For brevity, we omit the type annotation on dependent pairs, as in $\langle e_1, e_2 \rangle$. Note that let-bound definitions do not include type annotations; this is not standard, but type checking is still decidable (Severi & Poll, 1994), and it simplifies our ANF translation. As is standard, we assume uniqueness of names and consider syntax up to α-equivalence.

The following table summarizes the judgments for ECC and the new notation introduced, in particular the notation for *definitions* in contexts and *substitution* of variables for expressions.

Judgment	**Meaning**
$\vdash \Gamma$	The environment Γ is well formed
$\Gamma \vdash e : A$	The expression e has type A under environment Γ
$\Gamma \vdash A \preceq B$	The type A is a subtype of B under environment Γ
$\Gamma \vdash e \equiv e'$	The expressions e and e' are *judgmentally* equivalent under environment Γ
$\Gamma \vdash e \triangleright^* e'$	The expression e multi-steps to e' under environment Γ
$\Gamma \vdash e \triangleright e'$	The expression e single-steps to e' under environment Γ
Notation	**Meaning**
$\Gamma, x \overset{\delta}{=} e : A$	The variable x is defined to be e of type A (in some environment Γ)
$e'[x := e]$	Substitute the expression e for the variable x in the expression e'

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

In Figure 3, we define *definitional equivalence* (or just *equivalence*) $\Gamma \vdash e \equiv e'$ as conversion up to η-equivalence. We use the notation $e_1 \equiv e_2$ for equivalence, eliding the environment when it is obvious or unnecessary. We also define *cumulativity* (subtyping) $\Gamma \vdash A \preceq B$, to allow types in lower universes to inhabit higher universes.

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

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

4 Target: ECC with ANF support

Our target language, $\mathbf{CC}_e^A$, is a variant of ECC with a modified typing rule for dependent **if** that introduces propositional equalities between terms and equality reflection for accessing assumed equalities. The type system of $\mathbf{CC}_e^A$ is not particularly novel as its type theory is adapted from the extensional Calculus of Constructions (Oury, 2005). While $\mathbf{CC}_e^A$ supports ANF syntax, the full language is not ANF-restricted; it has the same syntax as ECC. We do not restrict the full language because maintaining ANF while type checking adds needless complexity, especially when checking for equality since conversion often breaks ANF. Instead, we show that our compiler generates only ANF restricted terms in $\mathbf{CC}_e^A$ and define a separate ANF-preserving machine-like semantics for evaluating programs in ANF. We typeset $\mathbf{CC}_e^A$ in a **bold, red, serif font**; in later sections, we reserve this font exclusively for the ANF restricted $\mathbf{CC}_e^A$. 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 can imagine the compilation process as either (1) generating ANF syntax in $\mathbf{CC}_e^A$ from ECC or (2) as first embedding ECC in $\mathbf{CC}_e^A$ and then rewriting $\mathbf{CC}_e^A$ terms into ANF. In Section 5, we present the compiler as process (1), a compiler from ECC to ANF $\mathbf{CC}_e^A$. In this section, we develop most of the supporting metatheory necessary for ANF as intra-language equivalences and process (2) may be a more helpful intuition.

We give the ANF syntax for $\mathbf{CC}_e^A$ in Figure 5(a). 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. We add the identity type $\mathbf{V} \equiv \mathbf{V}'$

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

$$
\begin{array}{rcll}
x & \triangleright_\delta & e & \text{where } x \overset{\delta}{=} e : A \in \Gamma \\
(\lambda\, x : A.\, e_1)\; e_2 & \triangleright_\beta & e_1[x := e_2] & \\
\mathsf{fst}\;\langle e_1, e_2\rangle & \triangleright_{\sigma_1} & e_1 & \\
\mathsf{snd}\;\langle e_1, e_2\rangle & \triangleright_{\sigma_2} & e_2 & \\
\mathsf{let}\; x = e_1\; \mathsf{in}\; e_2 & \triangleright_\zeta & e_2[x := e_1] & \\
\mathsf{if\;true\;then}\; e_1\; \mathsf{else}\; e_2 & \triangleright_{\mathbb{B}_1} & e_1 & \\
\mathsf{if\;false\;then}\; e_1\; \mathsf{else}\; e_2 & \triangleright_{\mathbb{B}_2} & e_2 & \\
\mathsf{indnat}\; A\; \mathsf{zero}\; e_1\; e_2 & \triangleright_{\iota_1} & e_1 & \\
\mathsf{indnat}\; A\; (\mathsf{succ}\; e)\; e_1\; e_2 & \triangleright_{\iota_2} & (e_2\; e)\; (\mathsf{indnat}\; A\; e\; e_1\; e_2) &
\end{array}
$$

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

$$
\cdots \qquad \frac{\Gamma, x \overset{\delta}{=} e : A \vdash e_1 \triangleright^* e_2}{\Gamma \vdash \mathsf{let}\; x = e\; \mathsf{in}\; e_1 \triangleright^* \mathsf{let}\; x = e\; \mathsf{in}\; e_2}\;[\textsc{Red-Cong-Let}]
$$

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

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

Fig. 2: ECC dynamic semantics (excerpt).

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\frac{\Gamma \vdash e_1 \triangleright^* e \qquad \Gamma \vdash e_2 \triangleright^* e}{\Gamma \vdash e_1 \equiv e_2}\,[\equiv]$$

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

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

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

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

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

Fig. 3: ECC equivalence and subtyping (excerpt).

and **refl V** to enable preserving dependent typing for **if** expressions and the join-point optimization, further described in Section 6. We do not require an elimination form for identity types; they are instead used via the restricted form of equality reflection in the equivalence judgment. A continuation **K** is a program with a hole and is composed **K**[**N**] with a computation **N** to form a configuration **M**. For example, (**let x** $=$ [$\cdot$] **in snd x**)[**N**] $=$ (**let x** $=$ **N in snd x**). Since continuations are not first-class objects, we cannot express control effects. Note that despite the *syntactic* distinctions, we still do not enforce a *phase* distinction—configurations (programs) can appear in types. Finally in Figure 5(b), we give the full non-ANF syntax, denoted by meta-variables **e**, **A**, and **B**. As done with ECC, we usually use the meta-variable **e** to evoke a term and the meta-variables **A** and **B** to evoke a type.

To summarize, the meaning of the judgments of CC_e^A is the same as ECC. However, CC_e^A has additional syntax for the identity type.

Judgment	**Meaning**
$\vdash \Gamma$	The environment Γ is well formed
$\Gamma \vdash e : A$	The expression **e** has type **A** under environment Γ
$\Gamma \vdash A \preceq B$	The type **A** is a subtype of **B** under environment Γ
$\Gamma \vdash e \equiv e'$	The expressions **e** and **e′** are *judgmentally* equivalent under Γ
$\Gamma \vdash e \triangleright^* e'$	The expression **e** multi-steps to **e′** under Γ
$\Gamma \vdash e \triangleright e'$	The expression **e** single-steps to **e′** under environment Γ
Notation	**Meaning**
$\Gamma, x \stackrel{\delta}{=} e : A$	The variable **x** is defined to be **e** of type **A** (in some environment Γ)
$e'[x := e]$	Substitute the expression **e** for the variable **x** in the expression **e′**
CC_e^A Syntax	Meaning
$p : e \equiv e'$	**p** has the identity type $e \equiv e'$, internalizing the judgment $\Gamma \vdash e \equiv e'$

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

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

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

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x : A \vdash B : \mathsf{Type}_i}{\Gamma \vdash \Pi\, x : A.\, B : \mathsf{Type}_i}\ [\textsc{Prod-Type}] \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\ e : B[x := \mathsf{fst}\ e]}\ [\textsc{Snd}]$$

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

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

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash e : B}\ [\textsc{Conv}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}_0}\ [\textsc{Nat}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}\ [\textsc{Zero}] \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ e : \mathsf{Nat}}\ [\textsc{Succ}]$$

$$\frac{\begin{array}{c}\Gamma, x : \mathsf{Nat} \vdash A : U \qquad \Gamma \vdash e : \mathsf{Nat} \\ \Gamma \vdash e_1 : A[x := \mathsf{zero}] \qquad \Gamma \vdash e_2 : \Pi\, n : \mathsf{Nat}.\, \Pi\, r : A[x := n].\, A[x := \mathsf{succ}\ n]\end{array}}{\Gamma \vdash \mathsf{indnat}\ A\ e\ e_1\ e_2 : A[x := e]}\ [\textsc{ElimNat}]$$

Fig. 4: ECC typing (excerpt).

We give the new typing rules in Figure 6. The rules for the identity type are standard. The key change in CC_e^A is in the typing rule for **if**. The typing rule for **if e then $\mathbf{e_1}$ else $\mathbf{e_2}$** introduces a propositional equality into the typing environment for each branch. These 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 type-preserving ANF transformation of **if**.

We require a new definition of equivalence to support extensionality, as shown in Figure 7. The rule [≡-Reflect] is used to determine two terms are equivalent if there is a proof of their equality. In particular, we use this rule to access equalities in the context such as those introduced in the typing rule for **if**. We add congruence rules for all syntactic forms including the identity type. The rules [≡-Subst][7] and [≡-Step] state that equivalence is preserved under substitution and the standard reduction defined for ECC. We also include η-equivalence as defined for ECC and η-equivalences for booleans.

To ensure reduction preserves ANF, we define composition of a continuation **K** and a configuration **M**, Figure 8, typically called *renormalization* in the literature (Sabry

[7] Formally, this rule is admissible. However, we make it explicit as we tried several various simplifications of extensionality in which it is not admissible and find it useful to mark this requirement. We discuss on in Section 7.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$$
\begin{array}{llll}
\textit{Universes} & \mathbf{U} & ::= & \mathbf{Prop} \mid \mathbf{Type}_i \\
\textit{Values} & \mathbf{V} & ::= & \mathbf{x} \mid \mathbf{U} \mid \boldsymbol{\lambda}\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \boldsymbol{\Pi}\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \\
 & & \mid & \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \langle \mathbf{V}, \mathbf{V} \rangle \mid \mathbf{Bool} \\
 & & \mid & \mathbf{true} \mid \mathbf{false} \mid \mathbf{Nat} \mid \mathbf{zero} \\
 & & \mid & \mathbf{succ}\ \mathbf{V} \mid \mathbf{refl}\ \mathbf{V} \mid \mathbf{V} \equiv \mathbf{V} \\
\textit{Computations} & \mathbf{N} & ::= & \mathbf{V} \mid \mathbf{V}\ \mathbf{V} \mid \mathbf{fst}\ \mathbf{V} \mid \mathbf{snd}\ \mathbf{V} \\
 & & \mid & \mathbf{indnat}\ \mathbf{M}\ \mathbf{V}\ \mathbf{V}\ \mathbf{V} \\
\textit{Configurations} & \mathbf{M} & ::= & \mathbf{N} \mid \mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{M} \\
 & & \mid & \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 \\
\textit{Continuations} & \mathbf{K} & ::= & [\cdot] \mid \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M} \\
\textit{Environments} & \boldsymbol{\Gamma} & ::= & \cdot \mid \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{V} \mid \boldsymbol{\Gamma}, \mathbf{x} \stackrel{\delta}{=} \mathbf{N} : \mathbf{N}
\end{array}
$$

(a) Run-time Syntax

$$
\begin{array}{lll}
\mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \mathbf{U} \mid \boldsymbol{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \\
 & \mid & \boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{e} \mid \mathbf{e}\ \mathbf{e} \\
 & \mid & \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \\
 & \mid & \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \\
 & \mid & \mathbf{fst}\ \mathbf{e} \mid \mathbf{snd}\ \mathbf{e} \\
 & \mid & \mathbf{let}\ \mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{Bool} \\
 & \mid & \mathbf{true} \mid \mathbf{false} \\
 & \mid & \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2 \\
 & \mid & \mathbf{Nat} \mid \mathbf{zero} \mid \mathbf{succ}\ \mathbf{e} \\
 & \mid & \mathbf{indnat}\ \mathbf{A}\ \mathbf{e}\ \mathbf{e}_1\ \mathbf{e}_2
\end{array}
$$

(b) Typing Syntax

Fig. 5: CC_e^A syntax.

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

...

$$
\frac{\boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{Bool} \qquad \begin{array}{c}\boldsymbol{\Gamma}, \mathbf{x} : \mathbf{Bool} \vdash \mathbf{B} : \mathbf{U} \\ \boldsymbol{\Gamma}, \mathbf{p} : \mathbf{e} \equiv \mathbf{true} \vdash \mathbf{e}_1 : \mathbf{B}[\mathbf{x} := \mathbf{true}] \qquad \boldsymbol{\Gamma}, \mathbf{p} : \mathbf{e} \equiv \mathbf{false} \vdash \mathbf{e}_2 : \mathbf{B}[\mathbf{x} := \mathbf{false}]\end{array}}{\boldsymbol{\Gamma} \vdash \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2 : \mathbf{B}[\mathbf{x} := \mathbf{e}]}\ [\textsc{If}]
$$

$$
\frac{\boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{A}}{\boldsymbol{\Gamma} \vdash \mathbf{refl}\ \mathbf{e} : \mathbf{e} \equiv \mathbf{e}}\ [\textsc{Refl}] \qquad \frac{\boldsymbol{\Gamma} \vdash \mathbf{A} : \mathbf{Type}_i \qquad \boldsymbol{\Gamma} \vdash \mathbf{A}' : \mathbf{Type}_i}{\boldsymbol{\Gamma} \vdash \mathbf{A} \equiv \mathbf{A}' : \mathbf{Type}_i}\ [\textsc{Equiv}]
$$

Fig. 6: CC_e^A typing (excerpt).

& Wadler, 1997; Kennedy, 2007). In ANF, all continuations are left associated, so the standard definition of substitution does not preserve ANF, unlike in alternatives such as CPS or monadic form. Note that β-reduction takes an ANF configuration $\mathbf{K}[(\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{M})\ \mathbf{V}]$ but would naïvely produce $\mathbf{K}[\mathbf{M}[\mathbf{x} := \mathbf{V}]]$. Substituting the term $\mathbf{M}[\mathbf{x} := \mathbf{V}]$, a *configuration*, into the continuation $\mathbf{K}$ could result in the non-ANF term $\mathbf{let}\ \mathbf{x} = \mathbf{M}\ \mathbf{in}\ \mathbf{M}'$. When composing a continuation with a configuration, $\mathbf{K}\langle\langle \mathbf{M} \rangle\rangle$, we unnest all continuations so they remain left associated. Note that these definitions rely on our uniqueness-of-names assumption.

Digression on composition in ANF. In the literature, the composition operation $\mathbf{K}\langle\langle \mathbf{M} \rangle\rangle$ is usually introduced as *renormalization*, as if the only intuition for why it exists is "well, it happens that ANF is not preserved under β-reduction." It is not mere coincidence; the intuition for this operation is composition, and having a syntax for composing terms is not only useful for stating β-reduction, but useful for all reasoning about ANF. This should not come as a surprise—compositional reasoning is useful. The only surprise is that the composition operation is not the usual one used in programming language semantics, that is, 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*

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\frac{\Gamma \vdash e : e_1 \equiv e_2}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Reflect}] \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \mathbf{refl}\ e \equiv \mathbf{refl}\ e'}\ [\equiv\text{-Cong-Refl}]$$

$$\frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash A' \equiv B'}{\Gamma \vdash (A \equiv A') \equiv (B \equiv B')}\ [\equiv\text{-Cong-Equiv}] \qquad \frac{\Gamma \vdash e \rhd e'}{\Gamma \vdash e \equiv e'}\ [\equiv\text{-Step}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2}{\Gamma \vdash e[x := e_1] \equiv e[x := e_2]}\ [\equiv\text{-Subst}]$$

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

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

$$\frac{\Gamma \vdash e \equiv \mathbf{true}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \equiv e_1}\ [\equiv\text{-If-}\beta_1] \qquad \frac{\Gamma \vdash e \equiv \mathbf{false}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \equiv e_2}\ [\equiv\text{-If-}\beta_2]$$

$$\frac{}{\Gamma \vdash \mathbf{if}\ e'\ \mathbf{then}\ e\ \mathbf{else}\ e \equiv e}\ [\equiv\text{-If2}] \qquad \ldots$$

Fig. 7: CC_e^A equivalence (excerpt).

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

$$\begin{aligned} K\langle\!\langle N \rangle\!\rangle &\overset{\text{def}}{=} K[N] \\ K\langle\!\langle \mathbf{let}\ x = N'\ \mathbf{in}\ M \rangle\!\rangle &\overset{\text{def}}{=} \mathbf{let}\ x = N'\ \mathbf{in}\ K\langle\!\langle M \rangle\!\rangle \\ K\langle\!\langle \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \rangle\!\rangle &\overset{\text{def}}{=} \mathbf{if}\ V\ \mathbf{then}\ K\langle\!\langle M_1 \rangle\!\rangle\ \mathbf{else}\ K\langle\!\langle M_2 \rangle\!\rangle \end{aligned}$$

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

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

Fig. 8: Composition of configurations.

or *configuration*. That would not be well typed. So how do we compose computations with configurations? We can use $\mathbf{let}$, as in $\mathbf{let}\ y = N\ \mathbf{in}\ M$, 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 $\mathbf{let}\ y = [\cdot]\ \mathbf{in}\ M'$, since this would result in the non-ANF (but valid monadic) expression $\mathbf{let}\ y = M\ \mathbf{in}\ M'$. 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.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$\boxed{\mathbf{M} \mapsto \mathbf{M}'}$

$$\begin{array}{rcl}
\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M})\ \mathbf{V}] & \mapsto_\beta & \mathbf{K}\langle\!\langle \mathbf{M}[\mathbf{x} := \mathbf{V}] \rangle\!\rangle \\
\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] & \mapsto_{\sigma_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{snd}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] & \mapsto_{\sigma_2} & \mathbf{K}[\mathbf{V}_2] \\
\mathbf{K}[\mathbf{indnat}\ \mathbf{M}\ \mathbf{zero}\ \mathbf{V}_1\ \mathbf{V}_2] & \mapsto_{\iota_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{indnat}\ \mathbf{M}\ (\mathbf{succ}\ \mathbf{V})\ \mathbf{V}_1\ \mathbf{V}_2] & \mapsto_{\iota_2} & \mathbf{let}\ \mathbf{x}_1 = (\mathbf{V}_2\ \mathbf{V})\ \mathbf{in}\ \mathbf{let}\ \mathbf{x}_2 = (\mathbf{indnat}\ \mathbf{M}\ \mathbf{V}\ \mathbf{V}_1\ \mathbf{V}_2)\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2] \\
\mathbf{let}\ \mathbf{x} = \mathbf{V}\ \mathbf{in}\ \mathbf{M} & \mapsto_\zeta & \mathbf{M}[\mathbf{x} := \mathbf{V}] \\
\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 & \mapsto_{\mathbb{B}_1} & \mathbf{M}_1 \\
\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 & \mapsto_{\mathbb{B}_2} & \mathbf{M}_2
\end{array}$$

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

$$\frac{}{\mathbf{M} \mapsto^* \mathbf{M}}\ [\textsc{Red-Refl}] \qquad \frac{\mathbf{M} \mapsto \mathbf{M}_1 \qquad \mathbf{M}_1 \mapsto^* \mathbf{M}'}{\mathbf{M} \mapsto^* \mathbf{M}'}\ [\textsc{Red-Trans}]$$

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

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

Fig. 9: $\mathrm{CC}_e^{\mathcal{A}}$ evaluation.

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

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

Fig. 10: $\mathrm{CC}_e^{\mathcal{A}}$ continuation typing.

In Figure 9, we present the call-by-value (CBV) evaluation semantics for ANF $\mathrm{CC}_e^{\mathcal{A}}$ terms. The semantics is only for the run-time evaluation of configurations; during type checking, we continue to use the conversion relation defined in Section 3. The semantics is essentially standard, but recall that β-reduction produces a configuration $\mathbf{M}$ which must be composed with the existing continuation $\mathbf{K}$. To maintain ANF, the natural number eliminator for the $\mathbf{succ}$ case must first let-bind the intermediate application and recursive call before plugging the final application into the existing continuation $\mathbf{K}$.

4.1 Dependent continuation typing

The ANF translation manipulates continuations $\mathbf{K}$ as independent entities. To reason about them, and thus to reason about the translation, we introduce continuation typing, defined in Figure 10. The type $(\mathbf{M}' : \mathbf{A}) \Rightarrow \mathbf{B}$ of a continuation expresses that this continuation expects to be composed with a term equal to the configuration $\mathbf{M}'$ of type $\mathbf{A}$ and returns a result of type $\mathbf{B}$ when completed. Normally, $\mathbf{M}'$ is equivalent to some computation $\mathbf{N}$, but it must be generalized to a configuration $\mathbf{M}'$ to support typing $\mathbf{if}$ expressions. This type formally expresses the idea that $\mathbf{M}'$ is depended upon in the rest of the computation. The [K-Bind]

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

rule ensures we have access to the specific $\mathbf{M}'$ needed to type body of the **let** continuation and is similar to the rule [T-CONT] used in the type-preserving CPS translation by Bowman *et al*. (2018). The rule [K-EMPTY] has an arbitrary term $\mathbf{M}'$, since an empty continuation $[\cdot]$ 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 $\mathbf{K}[\mathbf{N}]$ (generally, for configurations $\mathbf{K}\langle\langle\mathbf{M}\rangle\rangle$) by composing the typing derivations for $\mathbf{K}$ and $\mathbf{N}$. To get this property, a continuation type must express not merely the *type* of its hole $\mathbf{A}$, but *which term* $\mathbf{N}$ will be bound in the hole. We see this formally from the typing rule [LET] (the same for CC_e^A as for ECC in Section 3), since showing that $\mathbf{let}\ \mathbf{y} = \mathbf{N}\ \mathbf{in}\ \mathbf{M}$ is well typed requires showing that $\mathbf{y} \stackrel{\delta}{=} \mathbf{N} : \mathbf{A} \vdash \mathbf{M}$, that is, requires knowing the definition $\mathbf{y} \stackrel{\delta}{=} \mathbf{N} : \mathbf{A}$. If we omit the expression $\mathbf{N}$ from the type of a continuation, we know there are some configurations $\mathbf{K}[\mathbf{N}]$ that we cannot type check *compositionally*. Intuitively, if all we knew about $\mathbf{y}$ 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* $\mathbf{N}$ into one that depends on an *arbitrary* $\mathbf{y}$. We prove that our continuation typing is compositional in this way, Lemma 4.1 (Continuation Cut).

Note that the result of a continuation type cannot depend on the term that will be plugged in for the hole, that is, for a continuation $\mathbf{K} : (\mathbf{M}' : \mathbf{A}) \Rightarrow \mathbf{B}$, $\mathbf{B}$ does not depend on $\mathbf{M}'$. 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 $\mathbf{K}'$. Since composing any continuation $\mathbf{K} : (\mathbf{M}' : \mathbf{A}) \Rightarrow \mathbf{B}$ with any continuation $\mathbf{K}'$ results in a new continuation with the final result type $\mathbf{B}$, then the composition of any two continuations cannot depend on the type of the hole.

To prove that continuation typing is not an extension to the type system—that is, is admissible—we prove Lemmas 4.1 and 4.3, that plugging a well-typed computation or configuration into a well-typed continuation results in a well-typed term of the expected type.

We first show Lemma 4.1 (Continuation Cut), which is simple. This lemma tells us that our continuation typing allows for compositional reasoning about configurations $\mathbf{K}[\mathbf{N}]$ whose result types do not depend on $\mathbf{N}$.

Lemma 4.1 (Continuation Cut). *If* $\mathbf{\Gamma} \vdash \mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ *and* $\mathbf{\Gamma} \vdash \mathbf{N} : \mathbf{A}$ *then* $\mathbf{\Gamma} \vdash \mathbf{K}[\mathbf{N}] : \mathbf{B}$.

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

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

Continuation typing seems to require that we compose a continuation $\mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ *syntactically* with $\mathbf{N}$, but we will need to compose with some $\mathbf{N}' \equiv \mathbf{N}$. It is preferable 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.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Lemma 4.2 (Continuation 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: $K = \mathbf{let}\ x = [\cdot]\ \mathbf{in}\ M'$

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

Note that anywhere in the derivation $\Gamma, x \overset{\delta}{=} N : A \vdash M' : B$ that $x \overset{\delta}{=} N : A$ 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 $\Gamma, x \overset{\delta}{=} N' : A \vdash M' : B$. ■

Some presentations of context typing (Gordon, 1995)[8], in non-dependent settings, use a rule like 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 expected term in the hole is well typed. Then, Lemma 4.1 (Continuation 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 any changes as the definition of continuations change.

The final lemma about continuation typing is also the key to why ANF is type preserving for **if**. The heterogeneous composition operations perform the ANF translation on **if** expressions by composing the branches with the current continuation, reassociating all intermediate computations. The following lemma states that if a configuration M is well typed and equivalent to another (well-typed) configuration M' under an extended environment Γ', and the continuation K is well typed with respect to M', then the composition $K\langle\langle M\rangle\rangle$ is well typed. We include the additional environment Γ', as this environment contains information about new variables introduced through the composition with M, such as definitions and propositional equalities. We choose to include an explicit extra environment Γ' to avoid needing to prove tedious properties about individual environments. Thus,

[8] This presentation of context typing goes back at least to Gordon (1995), and has seen much use in the literature on operational approaches to full abstraction, logical relations, and secure compilation. Pitts (1997) gives a summary of some of the early work and their application to program equivalence, particularly observation equivalence.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

the proof is simple, except for the **if** case, which essentially must prove that ANF is type preserving for **if**.

Lemma 4.3 (Heterogeneous Continuation Cut). *If* $\Gamma \vdash \mathbf{M} : \mathbf{A}$, $\Gamma, \Gamma' \vdash \mathbf{M}' : \mathbf{A}'$, *and* $\Gamma, \Gamma' \vdash \mathbf{K} : (\mathbf{M}' : \mathbf{A}') \Rightarrow \mathbf{B}$ *such that* $\Gamma, \Gamma' \vdash \mathbf{M} \equiv \mathbf{M}'$ *and* $\Gamma, \Gamma' \vdash \mathbf{A} \equiv \mathbf{A}'$, *then* $\Gamma, \Gamma' \vdash \mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle : \mathbf{B}$.

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

Case: $\Gamma \vdash \mathbf{N} : \mathbf{A}$, by Lemma 4.2.

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

Our goal follows by the induction hypothesis applied to the sub-expression $\mathbf{M}$ if we can show (1) $\mathbf{M} \equiv \mathbf{M}'$ and (2) $\mathbf{B}' \equiv \mathbf{A}'$ under a context $\mathbf{x} \stackrel{\delta}{=} \mathbf{N} : \mathbf{A}$.

(1) follows by the following equations and transitivity of $\equiv$:

$$\begin{aligned} \mathbf{M}' &\equiv \mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{M} && \text{by commutativity} \\ &\equiv \mathbf{M}[\mathbf{x} := \mathbf{N}] && \text{by } \rhd_\zeta \text{ and } [\equiv\text{-STEP}] \\ &\equiv \mathbf{M} && \text{by } \rhd_\delta\text{, since we have } \mathbf{x} \stackrel{\delta}{=} \mathbf{N} : \mathbf{A} \end{aligned}$$

For (2), note that $\mathbf{B}'[\mathbf{x} := \mathbf{N}] \equiv \mathbf{A}'$ from our premises.

$$\begin{aligned} \mathbf{A}' &\equiv \mathbf{B}'[\mathbf{x} := \mathbf{N}] && \text{by commutativity} \\ &\equiv \mathbf{B}' && \text{by } \rhd_\delta \text{ since we have } \mathbf{x} \stackrel{\delta}{=} \mathbf{N} : \mathbf{A} \end{aligned}$$

Case: $\Gamma \vdash \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 : \mathbf{B}'[\mathbf{x} := \mathbf{V}]$

These follow by the induction hypotheses applied to the sub-expressions $\mathbf{M}_1$ and $\mathbf{M}_2$ if we can show (1) $\mathbf{M}_1 \equiv \mathbf{M}'$ and (2) $\mathbf{B}'[\mathbf{x} := \mathbf{true}] \equiv \mathbf{A}'$ under a context $\mathbf{p} : \mathbf{V} \equiv \mathbf{true}$ (analogously for $\mathbf{M}_2$ and **false**).

For (1), note that $\mathbf{V} \equiv \mathbf{true}$ by [$\equiv$-REFLECT] and the typing rule [VAR].

$$\begin{aligned} \mathbf{M}_1 &\equiv \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 && \text{by } [\equiv\text{-IF-}\beta_1] \\ &\equiv \mathbf{M}' && \text{by premise} \end{aligned}$$

For (2), note that $\mathbf{B}'[\mathbf{x} := \mathbf{V}] \equiv \mathbf{A}'$ from our premises and weakening. By [$\equiv$-SUBST], we know $\mathbf{B}'[\mathbf{x} := \mathbf{true}] \equiv \mathbf{B}'[\mathbf{x} := \mathbf{V}]$ if $\mathbf{V} \equiv \mathbf{true}$, which follows by [$\equiv$-REFLECT] and the typing rule [VAR]. ■

4.2 Metatheory

To demonstrate that the new typing rule for **if** is consistent, we develop a syntactic model of CC_e^A in extensional CIC (eCIC). We also prove subject reduction for CC_e^A. Subject reduction is valuable property for a typed IL to model various optimizations through the equational theory of the language, such as modeling inlining as β-reduction. If subject reduction holds, then all these optimizations are type preserving.

4.2.1 Consistency

Our syntactic model essentially implements the new **if** rule using the *convoy pattern* (Chlipala, 2013), but leaves the rest of the propositional equalities and equality

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$[\![\mathbf{if\ e\ then\ e_1\ else\ e_2}]\!]_M \stackrel{\text{def}}{=} \begin{array}{l}(\text{if } [\![\mathbf{e}]\!]_M \text{ then } (\lambda\, p : \text{true} \equiv [\![\mathbf{e}]\!]_M.\ \text{subst (if-eta1 } [\![\mathbf{e_1}]\!]_M\ [\![\mathbf{e_2}]\!]_M\ p)\ [\![\mathbf{e_1}]\!]_M) \\ \qquad\qquad\quad \text{else } (\lambda\, p : \text{false} \equiv [\![\mathbf{e}]\!]_M.\ \text{subst (if-eta2 } [\![\mathbf{e_1}]\!]_M\ [\![\mathbf{e_2}]\!]_M\ p)\ [\![\mathbf{e_2}]\!]_M)) \\ \text{refl } [\![\mathbf{e}]\!]_M\end{array}$$

Auxiliary CIC definitions

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

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

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

Fig. 11: CC_e^A model in eCIC (excerpt).

reflection essentially unchanged. Each **if** expression $\mathbf{if\ e\ then\ e_1\ else\ e_2}$ is modeled as an **if** expression that returns a function expecting a proof that the predicate **e** is equal to **true** in the first branch and **false** in the second branch. The function, after receiving that proof, executes the code in the branch. The model **if** expression is then immediately applied to refl, the canonical proof of the identity type. We could eliminate equality reflection using the translation of Oury (2005), but this is not necessary for consistency.

The essence of the model is given in Figure 11. There is only one interesting rule, corresponding to our altered dependent if rule. The model relies on auxiliary definitions in CIC, including subst, if-eta1 and if-eta2, whose types are given as inference rules in Figure 11. Note that the model for CC_e^A's **if** is not valid ANF, so it does not suffice to merely *use* the convoy pattern if we want to take advantage of ANF for compilation.

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

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

Lemma 4.4 (Model Preserves Empty Type). $[\![\bot]\!]_M \equiv \bot$

The essence of showing both that equivalence is preserved and that typing is preserved is in showing that the auxiliary definitions in Figure 11 exist and are well typed. We give these definitions in Coq in the supplementary materials, and the equivalences hold and are well typed without any additional axioms.

Lemma 4.5 (Model Preserves Equivalence). *If* $\mathbf{e_1} \equiv \mathbf{e_2}$ *then* $[\![\mathbf{e_1}]\!]_M \equiv [\![\mathbf{e_2}]\!]_M$.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Lemma 4.6 (Model Preserves Typing). *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!]_M \vdash [\![e]\!]_M : [\![A]\!]_M$.

Consistency tells us that there does not exist a closed term of the empty type $\bot$. This allows us to interpret types as specifications and well-typed terms as proofs.

Theorem 4.7 (Consistency). *There is no* e *such that* $\cdot \vdash e : \bot$

4.2.2 Syntactic metatheory

CC_e^A satisfies all the usual syntactic metatheory properties, but the main property of interest for developing our typed IL is subject reduction. We present the proof of subject reduction, which follows the structure presented by Luo (1990).

Theorem 4.8 (Subject Reduction). *If* $\Gamma \vdash e : A$ *and* $\Gamma \vdash e \triangleright^* e'$, *then* $\Gamma \vdash e' : A$.

Proof By induction on $\Gamma \vdash e \triangleright^* e'$. ∎

The proof of subject reduction relies on some additional lemmas. In particular, the proof relies on subject reduction for single step reduction (in the [Red-Trans] case), and a lemma called Luo calls *context replacement* (in the [Red-Cong-Lam] and [Red-Cong-Let] cases).

Lemma 4.9 (Single Step Subject Reduction). *If* $\Gamma \vdash e : A$ *and* $\Gamma \vdash e \triangleright e'$, *then* $\Gamma \vdash e' : A$.

Proof By cases on $\Gamma \vdash e \triangleright e'$. Most cases are straightforward, except for $\triangleright_\beta$ and $\triangleright_\zeta$, which rely on Lemmas 4.12 and 4.13, respectively. ∎

Lemma 4.10 (Context Replacement). *If* $\Gamma, x : A, \Gamma' \vdash e : C$ *and* $\Gamma \vdash B \preceq A$, *then* $\Gamma, x : B, \Gamma' \vdash e : C$.

Proof By (mutual) induction on $\Gamma, x : A, \Gamma' \vdash e : C$. ∎

Because our typing contexts also contain definitions, we must also be able to replace a definition expression with an equivalent expression. This is required specifically for the [Red-Cong-Let] case in the proof of subject reduction.

Lemma 4.11 (Context Definition Replacement). *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 : C$, $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\Gamma, x \overset{\delta}{=} e' : A, \Gamma' \vdash e_1 : C$.

Proof By (mutual) induction on $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 : C$. ∎

Finally, the proof of single step subject reduction relies on a lemma Luo calls Cut, named after the cut rule in sequent calculus. We must also prove an additional similar lemma due to definitions in our context, specifically for the $\triangleright_\zeta$ case.

Lemma 4.12 (Cut). *If* $\Gamma, x : A, \Gamma' \vdash e' : C$ *and* $\Gamma \vdash e : A$, *then* $\Gamma, \Gamma'[x := e] \vdash e'[x := e] : C[x := e]$.

Proof By (mutual) induction on $\Gamma, x : A, \Gamma' \vdash e' : C$. ∎

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Lemma 4.13 (Cut With Definitions). *If* $\mathbf{\Gamma}, \mathbf{x} \stackrel{\delta}{=} \mathbf{e} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{e}' : \mathbf{C}$, *then* $\mathbf{\Gamma}, \mathbf{\Gamma}'[\mathbf{x} := \mathbf{e}] \vdash \mathbf{e}'[\mathbf{x} := \mathbf{e}] : \mathbf{C}[\mathbf{x} := \mathbf{e}]$.

Proof By (mutual) induction on $\mathbf{\Gamma}, \mathbf{x} \stackrel{\delta}{=} \mathbf{e} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{e}' : \mathbf{C}$. ■

The proofs of both versions of context replacement and Cut are done by mutual induction, because typing in CC_e^A is mutually defined with well-formedness of contexts. Typing also relies on subtyping in the [Conv] case, subtyping relies on equivalence, and equivalence now relies on typing due to the equality reflection rule [≡-Reflect]. The mutual reliance of all these judgments requires that these lemmas be proven by mutual induction. The lemma statements above are not the full statement of the lemma, which we omit for brevity, but the corollaries are of interest. For example, the full lemma definition for Lemma 4.10 is as follows:

Lemma 4.14 (Context Replacement (full definition)).

1. *If* $\vdash \mathbf{\Gamma}, \mathbf{x} : \mathbf{A}, \mathbf{\Gamma}'$ *and* $\mathbf{\Gamma} \vdash \mathbf{B} \preceq \mathbf{A}$, *then* $\vdash \mathbf{\Gamma}, \mathbf{x} : \mathbf{B}, \mathbf{\Gamma}'$.
2. *If* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{e} : \mathbf{C}$ *and* $\mathbf{\Gamma} \vdash \mathbf{B} \preceq \mathbf{A}$, *then* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{B}, \mathbf{\Gamma}' \vdash \mathbf{e} : \mathbf{C}$.
3. *If* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{C} \preceq \mathbf{C}'$ *and* $\mathbf{\Gamma} \vdash \mathbf{B} \preceq \mathbf{A}$, *then* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{B}, \mathbf{\Gamma}' \vdash \mathbf{C} \preceq \mathbf{C}'$.
4. *If* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{e} \equiv \mathbf{e}'$ *and* $\mathbf{\Gamma} \vdash \mathbf{B} \preceq \mathbf{A}$, *then* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{B}, \mathbf{\Gamma}' \vdash \mathbf{e} \equiv \mathbf{e}'$.
5. *If* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{e} \triangleright^* \mathbf{e}'$ *and* $\mathbf{\Gamma} \vdash \mathbf{B} \preceq \mathbf{A}$, *then* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{B}, \mathbf{\Gamma}' \vdash \mathbf{e} \triangleright^* \mathbf{e}'$.
6. *If* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{A}, \mathbf{\Gamma}' \vdash \mathbf{e} \triangleright \mathbf{e}'$ *and* $\mathbf{\Gamma} \vdash \mathbf{B} \preceq \mathbf{A}$, *then* $\mathbf{\Gamma}, \mathbf{x} : \mathbf{B}, \mathbf{\Gamma}' \vdash \mathbf{e} \triangleright \mathbf{e}'$.

The full definitions for Lemmas 4.11, 4.12, and 4.13 are included in Appendix 1.

4.3 Correctness of ANF evaluation

In CC_e^A, we have an ANF evaluation semantics for run time and a separate definitional equivalence and reduction system for type checking. 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. When this happens, we wrap terms in a distinctive boundary such as $\mathcal{NN}(\mathbf{M}[\mathbf{x} := \mathbf{M}'])$ and $\mathcal{NN}(\mathbf{K}[\mathbf{M}])$. The boundary indicates the term is not normal, that is, not in A-normal form. The boundary is only meant to communicate with the reader; formally, $\mathcal{NN}(\mathbf{e}) = \mathbf{e}$.

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). Lemma 4.15 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 plugging the result into the hole of the continuation $\mathbf{K}$. Formally, this states that composing continuations in ANF is sound with respect to standard substitution. The proof follows by straightforward equational reasoning.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Lemma 4.15 (Naturality). $\mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle \equiv \mathcal{NN}(\mathbf{K}[\mathbf{M}])$

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

Case: $\mathbf{M} = \mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{M}'$
Must show that
$\mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{K}\langle\!\langle \mathbf{M}' \rangle\!\rangle \equiv \mathcal{NN}(\mathbf{K}[\mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{M}'])$.

$$
\begin{aligned}
& \mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{K}\langle\!\langle \mathbf{M}' \rangle\!\rangle \\
\equiv\ & \mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{K}[\mathbf{M}'] && \text{by induction} \\
\equiv\ & \mathcal{NN}(\mathbf{K}[\mathbf{M}'][\mathbf{x} := \mathbf{N}']) && \text{by } \rhd_\zeta \text{ and } [\equiv\text{-STEP}] \\
=\ & \mathcal{NN}(\mathbf{K}[\mathbf{M}'[\mathbf{x} := \mathbf{N}']]) && \text{by uniqueness of names} \\
\equiv\ & \mathcal{NN}(\mathbf{K}[\mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{M}']) && \text{by } \rhd_\zeta \text{ and } [\equiv\text{-STEP}]
\end{aligned}
$$

Case: $\mathbf{M} = \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2$
Must show that
$\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{K}\langle\!\langle \mathbf{M}_1 \rangle\!\rangle\ \mathbf{else}\ \mathbf{K}\langle\!\langle \mathbf{M}_2 \rangle\!\rangle \equiv \mathcal{NN}(\mathbf{K}[\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2])$.
This follows by induction if we can show
$\mathcal{NN}(\mathbf{K}[\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2]) \equiv \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathcal{NN}(\mathbf{K}[\mathbf{M}_1])\ \mathbf{else}\ \mathcal{NN}(\mathbf{K}[\mathbf{M}_2])$.
We show this by cases on $\mathbf{K}$.

Case: $\mathbf{K} = [\cdot]$ Trivial.

Case: $\mathbf{K} = \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M}$
Must show that
$\mathbf{let}\ \mathbf{x} = \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2\ \mathbf{in}\ \mathbf{M} \equiv \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M})\ \mathbf{else}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M})$.

$$
\begin{aligned}
& \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M})\ \mathbf{else}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M}) \\
\equiv\ & \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}[\mathbf{x} := \mathbf{M}_1]\ \mathbf{else}\ \mathbf{M}[\mathbf{x} := \mathbf{M}_2] && \text{by } \rhd_\zeta \text{ and } [\equiv\text{-STEP}] \\
\equiv\ & \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}[\mathbf{x} := \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2] && \text{by } [\equiv\text{-IF-}\beta] \text{ and } [\equiv\text{-SUBST}] \\
& \quad \mathbf{else}\ \mathbf{M}[\mathbf{x} := \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2] \\
\equiv\ & \mathbf{let}\ \mathbf{x} = (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2)\ \mathbf{in}\ (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}\ \mathbf{else}\ \mathbf{M}) && \text{by } \rhd_\zeta \text{ and } [\equiv\text{-STEP}] \\
\equiv\ & \mathbf{let}\ \mathbf{x} = (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2)\ \mathbf{in}\ \mathbf{M} && \text{by } [\equiv\text{-IF2}]
\end{aligned}
$$

■

Next we show that our ANF evaluation semantics are sound with respect to definitional equivalence. To do that, we first show that the small-step semantics are sound. Then, we show soundness of the evaluation function.

Lemma 4.16 (Small-step soundness). *If* $\mathbf{M} \mapsto \mathbf{M}'$ *then* $\mathbf{M} \equiv \mathbf{M}'$.

Proof By cases on $\mathbf{M} \mapsto \mathbf{M}'$. Most cases follow easily from the ECC reduction relation and congruence, except for $\mapsto_\beta$ which requires Lemma 4.15. We give representative cases.

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

$$
\begin{aligned}
& \mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\ \mathbf{M}_1)\ \mathbf{V}] \\
\rhd^*\ & \mathcal{NN}(\mathbf{K}[\mathbf{M}_1[\mathbf{x} := \mathbf{V}]]) && \text{by } \beta \text{ and congruence} && (4.1) \\
\equiv\ & \mathbf{K}\langle\!\langle \mathbf{M}_1[\mathbf{x} := \mathbf{V}] \rangle\!\rangle && \text{by Lemma 4.15} && (4.2)
\end{aligned}
$$

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Case: $\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] \mapsto_{\sigma_1} \mathbf{K}[\mathbf{V}_1]$
Must show that $\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] \equiv \mathbf{K}[\mathbf{V}_1]$, which follows by $\triangleright_{\sigma_1}$ and congruence. ■

Theorem 4.17 (Evaluation soundness). $\vdash \mathbf{eval}(\mathbf{M}) \equiv \mathbf{M}$

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

5 ANF translation

The ANF translation is presented in Figure 12. The translation is standard, defined inductively over syntax and indexed by a current continuation. The continuation is used when translating a value and is composed together "inside-out" the same way continuation composition is defined in Section 4. When translating a value such as x, $\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}$, or Type_i, we plug the value into the current continuation and recursively translate 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 and performs the rest of the computation.

We prove that the translation always produces syntax in ANF (Theorem 5.1). The proof is straightforward.

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

Our goal is to prove type preservation: if e is well typed in the source, then $[\![\mathsf{e}]\!]$ 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 the dependent type system relies on. The type judgment is defined mutually inductively only with well-formedness of environments; all other judgments relied on are defined inductively. We proceed top-down, starting from our main theorem, in order to motivate where each lemma comes into play. The full proof of type preservation is omitted for brevity but is included in Appendix 1.

Type-preservation is stated below. We do not prove it directly. Since the ANF translation is indexed by a continuation $\mathbf{K}$, we need a stronger induction hypothesis to reason about the type of the continuation $\mathbf{K}$.

Theorem 5.2 (Type Preservation). *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]$.

Proof By Lemma 5.3, it suffices to show that $[\![\Gamma]\!] \vdash [\cdot] : ([\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]) \Rightarrow [\![\mathsf{A}]\!]$, which follows by [K-Empty]. ■

To see why we need a stronger induction hypothesis, consider the $\mathsf{snd}\ \mathsf{e}$ case of ANF translation: $[\![\mathsf{snd}\ \mathsf{e}]\!]\ \mathbf{K} \stackrel{\text{def}}{=} [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x}]$. The induction hypothesis for Theorem 5.2 proves that $[\![\mathsf{e}]\!] : [\![\Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}]\!]$, but this cannot be used to show that the *translation* of e with the new continuation $\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x}]$ is well typed. We need the induction hypothesis to also include typing information for the new continuation that expects the translated sub-expression $[\![\mathsf{e}]\!]$. Note the new continuation also composes the original continuation $\mathbf{K}$ with

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$\boxed{\llbracket e \rrbracket\, \mathbf{K} = \mathbf{M}}$

$$
\begin{array}{rcl}
\llbracket e \rrbracket & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\, [\cdot] \\
\llbracket x \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{x}] \\
\llbracket \mathsf{Prop} \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{Prop}] \\
\llbracket \mathsf{Type}\ _i \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{Type}\ _i] \\
\llbracket \Pi\, x : A.\, B \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\boldsymbol{\Pi}\, \mathbf{x} : \llbracket A \rrbracket.\, \llbracket B \rrbracket] \\
\llbracket \lambda\, x : A.\, e \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\boldsymbol{\lambda}\, \mathbf{x} : \llbracket A \rrbracket.\, \llbracket e \rrbracket] \\
\llbracket e_1\ e_2 \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e_1 \rrbracket\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in} \\
 & & \quad \llbracket e_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]) \\
\llbracket \Sigma\, x : A.\, B \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\boldsymbol{\Sigma}\, \mathbf{x} : \llbracket A \rrbracket.\, \llbracket B \rrbracket] \\
\llbracket \langle e_1, e_2 \rangle\ \mathsf{as}\ A \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e_1 \rrbracket\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in} \\
 & & \quad \llbracket e_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in} \\
 & & \qquad \mathbf{K}[(\langle \mathbf{x}_1, \mathbf{x}_2 \rangle\ \mathbf{as}\ \llbracket A \rrbracket)]) \\
\llbracket \mathsf{fst}\ e \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{fst}\ \mathbf{x}] \\
\llbracket \mathsf{snd}\ e \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x}] \\
\llbracket \mathsf{let}\ x = e\ \mathsf{in}\ e' \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \llbracket e' \rrbracket\, \mathbf{K} \\
\llbracket \mathsf{Bool} \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{Bool}] \\
\llbracket \mathsf{true} \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{true}] \\
\llbracket \mathsf{false} \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{false}] \\
\llbracket \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in} \\
 & & \quad \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ (\llbracket e_1 \rrbracket\, \mathbf{K})\ \mathbf{else}\ (\llbracket e_2 \rrbracket\, \mathbf{K}) \\
\llbracket \mathsf{Nat} \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{Nat}] \\
\llbracket \mathsf{zero} \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{zero}] \\
\llbracket \mathsf{succ}\ e \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{succ}\ \mathbf{x}] \\
\llbracket \mathsf{indnat}\ A\ e\ e_1\ e_2 \rrbracket\, \mathbf{K} & \stackrel{\text{def}}{=} & \llbracket e \rrbracket\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in} \\
 & & \quad \llbracket e_1 \rrbracket\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in} \\
 & & \qquad \llbracket e_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in} \\
 & & \qquad\quad \mathbf{K}[\mathbf{indnat}\ \llbracket A \rrbracket\ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2]))) \\
\llbracket \cdot \rrbracket = \cdot & & \\
\llbracket \Gamma, x : A \rrbracket & \stackrel{\text{def}}{=} & \llbracket \Gamma \rrbracket, \mathbf{x} : \llbracket A \rrbracket \\
\left\llbracket \Gamma, x \stackrel{\delta}{=} e : A \right\rrbracket & \stackrel{\text{def}}{=} & \llbracket \Gamma \rrbracket, \mathbf{x} \stackrel{\delta}{=} \llbracket e \rrbracket : \llbracket A \rrbracket
\end{array}
$$

Fig. 12: Naïve ANF translation.

a target computation $\mathbf{snd}\ \mathbf{x}$. Intuitively, the composition $\mathbf{K}[\mathbf{snd}\ \mathbf{x}]$ is well typed because it appears in a context where $\mathbf{snd}\ \mathbf{x}$ is equivalent to the translation of the source expression $\llbracket \mathsf{snd}\ e \rrbracket$.

We abstract this pattern into Lemma 5.3, which takes both a well-typed source term and a well-typed continuation as input, just like the translation. The ANF translation takes an accumulator (the continuation $\mathbf{K}$) and builds up the translation in an accumulator as a procedure from values to ANF terms. The lemma builds up a proof of correctness as an accumulator as well. The accumulator is a *proposition* that if the computation it receives is

well typed, then composing the continuation with the computation is well typed. Formally, we phrase this as: if we start with a well-typed term e, and a well-typed continuation $\mathbf{K}$, then translating e with $\mathbf{K}$ results in a well-typed term. The continuation $\mathbf{K}$ expects a term that is the translation of the source expression directly, under an extended environment $\boldsymbol{\Gamma}'$. Intuitively, this extended environment $\boldsymbol{\Gamma}'$ contains information about new variables introduced through the ANF translation, such as definitions and propositional equivalences. This lemma and its proof are main contributions of this work.

Lemma 5.3.

1. *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]$.
2. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $[\![\Gamma]\!], \boldsymbol{\Gamma}' \vdash \mathbf{K} : ([\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]) \Rightarrow \mathbf{B}$*, then* $[\![\Gamma]\!], \boldsymbol{\Gamma}' \vdash [\![\mathsf{e}]\!]\ \mathbf{K} : \mathbf{B}$.

Proof The proof is by induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash \mathsf{e} : \mathsf{A}$. Cases [Ax-Prop], [Snd], [ElimNat], and [If] are given, as they are representative.

Note that for all sub-expressions, e' of type A', $[\![\mathsf{e}']\!] : [\![\mathsf{A}']\!]$ holds by instantiating the induction hypothesis with the empty continuation $[\cdot] : ([\![\mathsf{e}']\!] : [\![\mathsf{A}']\!]) \Rightarrow [\![\mathsf{A}']\!]$. The general structure of each case is similar; we proceed by induction on a sub-expression and prove the new continuation $\mathbf{K}'$ is well typed by applications of [K-Bind]. Proving the body of $\mathbf{K}'$ is well typed requires using Lemma 4.3. This requires proving that $\mathbf{K}'$ is composed with a configuration $\mathbf{M}$ equivalent to $[\![\mathsf{e}]\!]$, and showing their types are equivalent. To show $\mathbf{M}$ is equivalent to $[\![\mathsf{e}]\!]$, we use the Lemmas 5.4 and 4.15 to allow for composing configurations and continuations. Additionally, since several typing rules substitute sub-expressions into the type system, we use Lemma 5.5 in these cases to show the types of $\mathbf{M}$ and $[\![\mathsf{e}]\!]$ are equivalent. Each non-value proof case focuses on showing these two equivalences.

Case: [Ax-Prop]

We must show that $[\![\Gamma]\!], \boldsymbol{\Gamma}' \vdash \mathbf{K}[\mathbf{Prop}] : \mathbf{B}$. This follows from Lemma 4.2.

Case: [Snd] Following our general structure, we must show (1) $\mathbf{snd}\ \mathbf{x}_1 \equiv [\![\mathsf{snd}\ \mathsf{e}]\!]$ in an environment where $\mathbf{x}_1 \overset{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]$ and (2) $[\![\mathsf{B}']\!][\mathbf{x} := \mathbf{fst}\ [\![\mathsf{e}]\!]] \equiv [\![\mathsf{B}'[\mathsf{x} := \mathsf{fst}\ \mathsf{e}]]\!]$.

For goal (1), we focus on $[\![\mathsf{snd}\ \mathsf{e}]\!]$:

$$\begin{aligned}
&[\![\mathsf{snd}\ \mathsf{e}]\!] \overset{\text{def}}{=} [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_1) && \\
&= \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_1 \langle\!\langle [\![\mathsf{e}]\!] \rangle\!\rangle && \text{by Lemma 5.4} \\
&\equiv \mathsf{let}\ \mathsf{x}_1 = [\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{snd}\ \mathsf{x}_1 && \text{by Lemma 4.15} \\
&\rhd_\zeta \mathbf{snd}\ [\![\mathsf{e}]\!] && \\
&\rhd_\delta \mathbf{snd}\ \mathbf{x}_1 &&
\end{aligned}$$

For goal (2), since $[\![\mathsf{B}'[\mathsf{x} := \mathsf{fst}\ \mathsf{e}]]\!] \equiv [\![\mathsf{B}']\!][\mathbf{x} := [\![\mathsf{fst}\ \mathsf{e}]\!]]$ by Lemma 5.5, we focus on showing $\mathbf{fst}\ [\![\mathsf{e}]\!] \equiv [\![\mathsf{fst}\ \mathsf{e}]\!]$. Focusing on $[\![\mathsf{fst}\ \mathsf{e}]\!]$, we have:

$$\begin{aligned}
&[\![\mathsf{fst}\ \mathsf{e}]\!] \overset{\text{def}}{=} [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x}_1) && \\
&= \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x}_1 \langle\!\langle [\![\mathsf{e}]\!] \rangle\!\rangle && \text{by Lemma 5.4} \\
&\equiv \mathsf{let}\ \mathsf{x}_1 = [\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{fst}\ \mathsf{x}_1 && \text{by Lemma 4.15} \\
&\rhd_\zeta \mathbf{fst}\ [\![\mathsf{e}]\!] &&
\end{aligned}$$

Case: [ElimNat] Following our general structure, we must show (1) $\mathbf{indnat}\ [\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2 \equiv [\![\mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2]\!]$ in an environment where

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$\mathbf{y} \overset{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Nat}]\!]$, $\mathbf{x_1} \overset{\delta}{=} [\![\mathsf{e}_1]\!] : [\![\mathsf{A}[\mathsf{zero} := \mathsf{x}]]\!]$, $\mathbf{x_2} \overset{\delta}{=} [\![\mathsf{e}_2]\!] : [\![\Pi\, \mathsf{n} : \mathsf{Nat}.\, \Pi\, \mathsf{x}' : \mathsf{A}[\mathsf{x} := \mathsf{n}].\, \mathsf{A}[\mathsf{x} := \mathsf{succ}\ \mathsf{n}]]\!]$ and (2) $[\![\mathsf{A}]\!][\mathbf{x} := \mathbf{y}] \equiv [\![\mathsf{A}[\mathsf{x} := \mathsf{e}]]\!]$.
For goal (1), we focus on $[\![\mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2]\!]$:

$$
\begin{aligned}
& [\![\mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2]\!] \\
& \overset{\text{def}}{=} [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2}))) \\
& = \mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2}))\langle\!\langle [\![\mathsf{e}]\!] \rangle\!\rangle \\
& \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ \mathsf{y} = [\![\mathsf{e}]\!]\ \mathsf{in}\ [\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2})) \\
& \text{by Lemma 4.15} \\
& \rhd_\zeta [\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ \mathbf{x_1}\ \mathbf{x_2})) \\
& = \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ \mathbf{x_1}\ \mathbf{x_2})\langle\!\langle [\![\mathsf{e}_1]\!] \rangle\!\rangle \\
& \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ \mathsf{x}_1 = [\![\mathsf{e}_1]\!]\ \mathsf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ \mathbf{x_1}\ \mathbf{x_2}) \\
& \text{by Lemma 4.15} \\
& \rhd_\zeta [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ [\![\mathsf{e}_1]\!]\ \mathbf{x_2}) \\
& = \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ [\![\mathsf{e}_1]\!]\ \mathbf{x_2}\langle\!\langle [\![\mathsf{e}_2]\!] \rangle\!\rangle \\
& \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ \mathsf{x}_2 = [\![\mathsf{e}_2]\!]\ \mathsf{in}\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ [\![\mathsf{e}_1]\!]\ \mathbf{x_2} \\
& \text{by Lemma 4.15} \\
& \rhd_\zeta\ \mathbf{indnat}\ [\![\mathsf{A}]\!]\ [\![\mathsf{e}]\!]\ [\![\mathsf{e}_1]\!]\ [\![\mathsf{e}_2]\!]
\end{aligned}
$$

For goal (2), since $[\![\mathsf{A}[\mathsf{x} := \mathsf{e}]]\!] \equiv [\![\mathsf{A}]\!][\mathbf{x} := [\![\mathsf{e}]\!]]$ by Lemma 5.5, and $\mathbf{y} \equiv [\![\mathsf{e}]\!]$ by the definition $\mathbf{y} \overset{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Nat}]\!]$, we conclude using [≡-Subst].

Case: [If] Following our general structure, we must show (1) $[\![\mathsf{e}_1]\!] \equiv [\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]$ in an environment where $\mathbf{x} \overset{\delta}{=} [\![\mathsf{e}]\!] : \mathbf{Bool}, \mathbf{p} : \mathbf{x} \equiv \mathbf{true}$ and (2) $[\![\mathsf{B}'[\mathsf{x}' := \mathsf{e}]]\!] \equiv [\![\mathsf{B}']\!][\mathbf{x}' := \mathbf{true}]$ (and analogously for e_2 where $\mathbf{x} \overset{\delta}{=} [\![\mathsf{e}]\!] : \mathbf{Bool}, \mathbf{p} : \mathbf{x} \equiv \mathbf{false}$).
Focusing on $[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]$ in goal (1), we have:

$$
\begin{aligned}
& [\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!] \\
& \overset{\text{def}}{=} [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!]) \\
& = \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!]\ \langle\!\langle [\![\mathsf{e}]\!] \rangle\!\rangle && \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ \mathsf{x} = [\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{if}\ \mathsf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!] && \text{by Lemma 4.15} \\
& \rhd_\zeta\ \mathbf{if}\ [\![\mathsf{e}]\!]\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!] \\
& \equiv [\![\mathsf{e}_1]\!] && \text{by [}{\equiv}\text{-If-}\beta_1\text{] since } [\![\mathsf{e}]\!] \equiv \mathbf{true} \text{ by [}{\equiv}\text{-Reflect] and [Var]}
\end{aligned}
$$

For goal (2), $[\![\mathsf{B}']\!][\mathbf{x}' := [\![\mathsf{e}]\!]] \equiv [\![\mathsf{B}']\!][\mathbf{x}' := \mathbf{true}]$ by Lemma 5.5. Since $[\![\mathsf{e}]\!] \equiv \mathbf{true}$ we conclude with [≡-Subst]. ■

Key steps in the proof require reasoning about equivalence of terms. To prove equivalence of terms, we need to know their syntax so a structural equivalence rule applies, which is not true of terms of the form $[\![\mathsf{e}]\!]\ \mathbf{K}$. To reason about this term, we need to show an equivalence between the compiler and ANF-composition, so we can reason instead about $\mathbf{K}\langle\!\langle [\![\mathsf{e}]\!] \rangle\!\rangle$, a definition we can unroll and we know is correct with respect to evaluation via Lemma 4.15 (Naturality). We prove this equivalence, Lemma 5.4 next. This essentially

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

tells us that our compiler is compositional, that is, respects separate compilation and composition in the target language. We can either first translate a program e under continuation $\mathbf{K}$ and then compose it with a continuation $\mathbf{K}'$, or we can first compose the continuations $\mathbf{K}$ and $\mathbf{K}'$ and then translate e under the composed continuation.

Lemma 5.4 (Compositionality). $\mathbf{K}'\langle\!\langle\, [\![\mathsf{e}]\!]\ \mathbf{K}\rangle\!\rangle = [\![\mathsf{e}]\!]\ \mathbf{K}'\langle\!\langle\mathbf{K}\rangle\!\rangle$.

Proof By induction on the structure of e. All value cases are trivial. The cases for non-values are all similar, following by definition of composition for continuations or configurations. ■

Corollary 5.4.1. $\mathbf{K}\langle\!\langle\, [\![\mathsf{e}]\!]\,\rangle\!\rangle = [\![\mathsf{e}]\!]\ \mathbf{K}$.

Similarly, at times we need to reason about the translation of terms or types that have a variable substituted, such as $[\![\mathsf{A}[\mathsf{x} := \mathsf{e}']]\!]$. We often use induction on a judgment $\Gamma \vdash \mathsf{e} : \mathsf{A}$, which gives us the structure of the expression e but not much of the structure of the type A. The type A may have a term substituted, as $[\![\mathsf{A}[\mathsf{x} := \mathsf{e}']]\!]$, but we have no information about the structure of A itself. Since we cannot reason about A directly, we cannot obtain the final term after substitution, and thus, we do not know the syntax of the compilation of this arbitrary term. We show another kind of compositionality with respect to this substitution, Lemma 5.5, which shows that substitution is equivalent to composing via continuations. Since standard substitution does not preserve ANF, this lemma does not equate terms in ANF, but CC_e^A terms that are not normal. We again mark this shift with the boundary term $\mathcal{NN}()$.

Lemma 5.5 (Substitution). $[\![\mathsf{e}[\mathsf{x} := \mathsf{e}']]\!]\ \mathbf{K} \equiv \mathcal{NN}(([\![\mathsf{e}]\!]\ \mathbf{K})[\mathbf{x} := [\![\mathsf{e}']\!]])$.

Our type system relies on a subtyping judgment, so we must show subtyping is preserved. The proof is uninteresting, except insofar as it is simple, while it seems to be impossible in prior work for CPS translation (Bowman *et al.*, 2018).

Lemma 5.6. *If* $\Gamma \vdash \mathsf{e} \preceq \mathsf{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \preceq [\![\mathsf{e}']\!]$.

Subtyping relies on type equivalence (but not vice versa), so we must also show the equivalence judgment is preserved. This lemma is also useful as a kind of compiler correctness property, ensuring that our notion of program equivalence (since types and terms are the same) is preserved through compilation.

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

Since equivalence is defined in terms of reduction and conversion, we must also show reduction and conversion are preserved up to equivalence in the target language. This is convenient, since reduction and conversion also define the dynamic semantics of programs. These proofs correspond to a forward simulation proof up to target language equivalence and also imply compiler correctness. The proofs are straightforward; intuitively, ANF is just adding a bunch of ζ-reductions.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

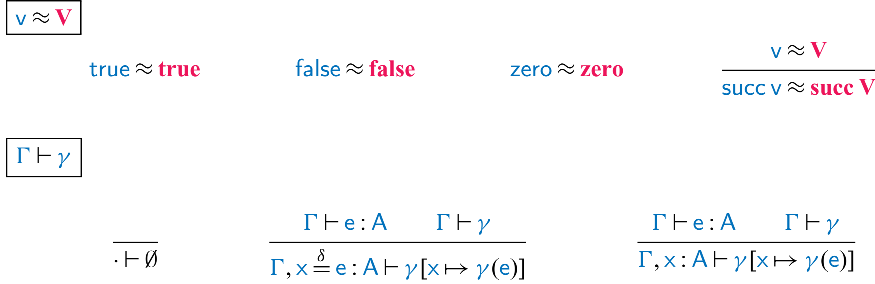

Fig. 13: Separate compilation definitions.

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

Lemma 5.9. *If* $\Gamma \vdash e \rhd^* e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$.

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

As a result of proving that ANF preserves the judgmental and syntactic structure that a dependent type system relies on, we can also easily prove correctness of separate compilation (with respect to the ANF evaluation semantics). To do this, we must define linking and a specification of when outputs are related across languages, independent of the compiler.

We define an independent specification relating observation across languages, which allows us to understand the correctness theorem without reading the compiler. We define the relation $v \approx \mathbf{V}$ to compare ground values in Figure 13.

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 13). Linking is defined by closing a term e such that $\Gamma \vdash e : A$ with a substitution $\Gamma \vdash \gamma$, written $\gamma(e)$. Any γ is valid for Γ if it maps each $x : A \in \Gamma$ to a closed term e of type A. For definitions in Γ, we require that if $x \overset{\delta}{=} e : A \in \Gamma$, then $\gamma[x \mapsto \gamma(e)]$, that is, the substitution must map x to a closed version of its definition e. We lift the ANF translation to substitutions.

Correctness of separate compilation says that we can either link then run a program in the source language semantics, that is, 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. The proof is straightforward from the compositionality properties and forward simulations we proved for type preservation.

Theorem 5.10 (Correctness of Separate Compilation). *If* $\Gamma \vdash e : A$*, (and* A *a base type) and* $\Gamma \vdash \gamma$ *then* $\mathbf{eval}([\![\gamma]\!]\,([\![e]\!])) \approx \mathsf{eval}(\gamma(e))$.

Proof The following diagram commutes, because $\equiv$ corresponds to $\approx$ on base types (nat or bool), the translation commutes with substitution, and preserves equivalence.

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

■

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\vdots$$

$$[\![\text{if e then e}_1\text{ else e}_2]\!]\ \mathbf{K} \stackrel{\text{def}}{=}$$
$$\mathbf{let\ f} = \boldsymbol{\lambda}\ \mathbf{y} : [\![B[x := e]]\!].\ \boldsymbol{\lambda}\ \mathbf{p} : \mathbf{y} \equiv [\![\text{if e then e}_1\text{ else e}_2]\!].\ \mathbf{K}[\mathbf{y}]$$
$$\mathbf{in}\ [\![e]\!]\ \mathbf{let\ x'} = [\cdot]$$
$$\mathbf{in\ if\ x'\ then}\ [\![e_1]\!]\ (\mathbf{let\ x_1} = [\cdot]\ \mathbf{in}\ (\mathbf{f\ x_1}\ (\mathbf{refl\ x_1})))$$
$$\mathbf{else}\ [\![e_2]\!]\ (\mathbf{let\ x_2} = [\cdot]\ \mathbf{in}\ (\mathbf{f\ x_2}\ (\mathbf{refl\ x_2})))$$
$$\text{where if e then e}_1\text{ else e}_2 : B[x := e]$$

Fig. 14: Join-point optimized ANF translation.

6 Join-point optimization

Recall from Figure 8 that the composition of a continuation $\mathbf{K}$ with an **if** configuration, $\mathbf{K}\langle\!\langle \mathbf{if\ V\ then\ M_1\ else\ M_2} \rangle\!\rangle$, duplicates $\mathbf{K}$ in the branches: $\mathbf{if\ V\ then\ K}\langle\!\langle \mathbf{M_1} \rangle\!\rangle\ \mathbf{else\ K}\langle\!\langle \mathbf{M_2} \rangle\!\rangle$. Similarly, the ANF translation in Figure 12 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.

CC_e^A supports implementing the join-point optimization, which avoids this code duplication. We modify the ANF translation from Section 5 to use the definition in Figure 14 and prove it is still correct and type preserving. Note that the new translation requires access to the type B from the derivation. We can do this either by defining the translation by induction over typing derivations or (preferably) modifying the syntax of if to include B as a type annotation, similar to dependent pairs in ECC or dependent case analysis in Coq.

Lemma 6.1.

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

Proof By induction.

Case: [IF] We proceed by induction on the branch sub-expressions and ensure their corresponding continuation is well typed. This means the application of the join point $\mathbf{f}$ must be on well-typed arguments of (1) $\mathbf{x_1} : [\![A[x := e]]\!]$ and (2) $\mathbf{refl\ x_1} : \mathbf{x_1} \equiv [\![\text{if e then e}_1\text{ else e}_2]\!]$ (analogously for $\mathbf{x_2}$ in the **false** branch). (1) follows from the equivalence $[\![A[x := e]]\!] \equiv [\![A[x := \text{true}]]\!]$, which has been shown before in Lemma 5.3. (2) follows if we show $(\mathbf{x_1} \equiv \mathbf{x_1}) \equiv (\mathbf{x_1} \equiv [\![\text{if e then e}_1\text{ else e}_2]\!])$ in an environment where $\mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![B'[x := \text{true}]]\!]$. By [≡-CONG-EQUIV], we must show $\mathbf{x_1} \equiv [\![\text{if e then e}_1\text{ else e}_2]\!]$, which by definition is $[\![e_1]\!] \equiv [\![\text{if e then e}_1\text{ else e}_2]\!]$, previously shown in Lemma 5.3. ∎

7 Future and related work

We first discuss possible extensions to the source language. We sketch adding recursion and inductive data types to our source language ECC and conjecture that our proof

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

technique can scale to these features. However, extending the source with extensional equality seems more difficult. We then discuss ways to efficiently type check CC_e^A terms constructed from the ANF translation. We conclude by discussing related work as alternatives to the ANF translation, in particular the CPS translation, Call-by-Push-Value, and monadic form. These alternatives either fail to scale to higher universes, or fail to compile to a language as “low-level” as CC_e^A in ANF.

7.1 Costs of type-preserving compilation

Compared to other approaches to solving linking errors, such as PCC (Necula, 1997) which requires finding and encoding witnesses of safety proofs, type-preserving compilation is a lightweight and effective technique (Morrisett *et al*., 1999; Xi & Harper, 2001; Shao *et al*., 2005).

Unfortunately, one major disadvantage is the compilation time. TIL (Tarditi *et al*., 1996), a type-preserving compiler for Standard ML, was found to be eight times slower than the non-type-preserving compiler SML/NJ. Chen *et al*. (2008) found that their type-preserving compiler for Microsoft’s Common Intermediate Language (CIL) was 82.8% slower than their base compiler, mostly due to type inference and writing type information to object files. We consider long compilation times a relatively small price to pay for the safety guarantees provided for the compiled program, particularly given the cost of unsafety (Stecklein *et al*., 2004; Briski *et al*., 2008).

It is also unclear that this cost is necessary. Shao *et al*. (1998) show that careful attention to representation of typed intermediate language nearly eliminated compile time overhead introduced in the FLINT compiler compared to the SML/NJ compiler. The focus in that work is with respect to large types, which is particularly relevant for dependent-type preservation.

However, annotations overhead with dependent types can be far larger. For example, annotation size (including both types and proofs) was 6x time larger than the code size in CompCert (Leroy, 2009), and one extension to CompCert doubled that (Stewart *et al*., 2015). If this sort of difference between annotation size and code size is typical, it could be a huge problem. It is not clear that is typical; Stewart *et al*. (2015) note that the doubling was in part due to duplication of definitions which should be unnecessary.

Removing inference from the type system could significantly improve compile time (Chen *et al*., 2008), but could increase code size and require more annotations. This time/space tradeoff has been studied in the context of LF (Necula & Rahul, 2001; Sarkar *et al*., 2005). Adapting these techniques to dependently typed languages in the style of Coq would likely be necessary for practical implementation of dependent-type preservation, but the issues are very similar to that of LF.

7.2 Recursive functions and match

Our source language ECC still lacks some key features of a general purpose dependently typed language. In particular, our language does not have a `fix` construct for defining recursive functions and a `match` construct for eliminating general inductive datatypes. These features require a *termination condition* to ensure termination of programs; thus, building

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

a type-preserving compiler for a language with these features would require the compiler preserve the termination condition as well. We could preserve a syntactic termination condition, as used by Coq; however, this requires a significantly more complex target guard condition. Bowman & Ahmed (2018) provide an example of preserving the guard condition through the closure conversion pass, where the guard condition must essentially become a data-flow analysis in order to track the flow of recursive calls through closures. This would greatly increase the complexity of the target type system and possibly affect type checking performance. This suggests that using a syntactic condition is undesirable for a dependently typed intermediate language. An alternative could be compiling this syntactic guard condition to sized types, but adding sized types to Coq is still a work in progress (Chan & Bowman, 2020).

However, assuming we have preserved the guard condition through the ANF translation with the aforementioned possible complexities, we conjecture that the technique presented in this paper scales to these language features. In particular, proving that the ANF translation of match is type preserving would be very similar to the technique used for dependent if. Suppose we have added a syntactic form $\mathsf{match}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ (\mathsf{A}\ \mathsf{z})\ \mathsf{return}\ \mathsf{B}\ \mathsf{with}\{(\mathsf{C}_i\ \mathsf{y}_i) \to \mathsf{e}_i\}$ and typing rule:

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A}\ \mathsf{e}' \qquad \Gamma, \mathsf{z} : \mathsf{A}', \mathsf{x} : \mathsf{A}\ \mathsf{z} \vdash \mathsf{B} : \mathsf{U} \qquad \Gamma, \mathsf{y}_i : \mathsf{B}_i \vdash \mathsf{e}_i : \mathsf{B}[\mathsf{z} := \mathsf{e}'_i][\mathsf{x} := \mathsf{C}_i\ \mathsf{y}_i]}{\Gamma \vdash \mathsf{match}\ \mathsf{e}\ \mathsf{as}\ \mathsf{x}\ \mathsf{in}\ (\mathsf{A}\ \mathsf{z})\ \mathsf{return}\ \mathsf{B}\ \mathsf{with}\{(\mathsf{C}_i\ \mathsf{y}_i) \to \mathsf{e}_i\} : \mathsf{B}[\mathsf{z} := \mathsf{e}'][\mathsf{x} := \mathsf{e}]}$$

The target language would have a similar syntactic $\mathbf{match}$ construct as a configuration $\mathbf{M}$. Any proofs of lemmas over configurations $\mathbf{M}$ should be updated to include the $\mathbf{match}$ construct. The ANF translation would first translate the scrutinee e and then push $\mathbf{K}$ into each branch of the $\mathbf{match}$. In order to prove this translation type preserving, we would change the target typing rule for $\mathbf{match}$ to include a propositional equality when checking each branch. That is, the premises $\Gamma, \mathsf{y}_i : \mathsf{B}_i \vdash \mathsf{e}_i : \mathsf{B}[\mathsf{z} := \mathsf{e}'_i][\mathsf{x} := \mathsf{C}_i\ \mathsf{y}_i]$ in the source match rule above would change to $\mathbf{\Gamma}, \mathbf{y}_i : \mathbf{B}_i, \mathbf{p} : \mathbf{e} \equiv \mathbf{C}_i\ \mathbf{y}_i \vdash \mathbf{e}_i : \mathbf{B}[\mathbf{z} := \mathbf{e}'_i][\mathbf{x} := \mathbf{C}_i\ \mathbf{y}_i]$. The target typing rule threads the equality $\mathbf{p} : \mathbf{e} \equiv \mathbf{C}_i\ \mathbf{y}_i$ when checking each branch, to record that the scrutinee $\mathbf{e}$ has reduced to a particular case of the match $\mathbf{C}_i\ \mathbf{y}_i$.

Threading an equality into the branches proves type preservation in the same way as with if statements with the equality $\mathbf{p} : \mathbf{e} \equiv \mathbf{true}$ (or $\mathbf{p} : \mathbf{e} \equiv \mathbf{false}$). The continuation $\mathbf{K}$ originally expects something of type $[\![\mathsf{B}[\mathsf{z} := \mathsf{e}'][\mathsf{x} := \mathsf{e}]]\!]$ but then is pushed into the branches and translated with something of type $[\![\mathsf{B}[\mathsf{z} := \mathsf{e}'_i][\mathsf{x} := \mathsf{C}_i\ \mathsf{y}_i]]\!]$. The equality $\mathbf{p} : \mathbf{e} \equiv \mathbf{C}_i\ \mathbf{y}_i$ in the context introduced by the target typing rule helps resolve the discrepancy between the substitution $[\mathsf{x} := \mathsf{e}]$ and $[\mathsf{x} := \mathsf{C}_i\ \mathsf{y}_i]$.

Assuming the termination condition can be preserved through the ANF translation, proving that ANF is type-preserving for fix would be similar to proving type preservation for functions. Given the typing rule for fix and the ANF translation:

$$\frac{\Gamma, \mathsf{f} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B} \qquad guard(\mathsf{f}, \mathsf{x}, \mathsf{e})}{\Gamma \vdash \mathsf{fix}\,\mathsf{f}(\mathsf{x} : \mathsf{A}).\,\mathsf{e} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}}$$

$$[\![\mathsf{fix}\,\mathsf{f}(\mathsf{x} : \mathsf{A}).\,\mathsf{e}]\!] \stackrel{\text{def}}{=} \mathbf{K}[\mathbf{fix}\,\mathbf{f}(\mathbf{x} : [\![\mathsf{A}]\!]).\,[\![\mathsf{e}]\!]]$$

We could show that the ANF translation is type preserving by Lemma 4.2 and show that the target fix expression $\mathbf{fix}\,\mathbf{f}(\mathbf{x} : [\![\mathsf{A}]\!]).\,[\![\mathsf{e}]\!]$ is well typed. This is easy to show using the target [Fix] typing rule, assuming the guard condition is preserved, and by induction on e.

7.3 Preserving extensional equality

A natural question is whether we can preserve extensional equality through ANF translation. Since the target language already supports this, it seems trivial to preserve, no?

Our study so far suggests that adding new features with the same pattern of dependency and judgmental structural does not take much effort. Adding new positive or negative types is not significantly more complicated if the translation already supports positive and negative types. Adding recursion does not seem difficult since the dependency is not new, although it adds a new judgment that must be preserved.

By contrast, extensional equality is a radical change to the judgmental structure of the source language. Currently, in our source language, the typing and equivalence judgments are separately defined—typing depends on equivalence, but not vice versa. Adding extensional equality typically makes these two judgment mutually dependent. This completely changes the structure of the type preservation proof—as we discussed in Section 5, we stage our proof by first showing reduction is preserved, then equivalence, etc, and finally that typing is preserved. This structure follows the structure of the source typing derivations. Because adding extensional equality changes the structure of source typing derivations, this entire proof could no longer be staged and would need to be mutually inductive.

Past authors have not had much success in solving these large mutual inductive proofs and instead avoid them. Barthe *et al*. (1999) discuss this problem as choose to use a Curry-style type system to avoid the problem. Bowman *et al*. (2018) discuss it also, but choose to use untyped equivalence and untyped reduction to avoid it. We follow the latter approach, but that is not tenable if we extend the source language with extensional equability.

We have considered alternative variants of extensional equality. For example, consider the following idea for a simplification of the Rule [≡-Reflect] which avoids mutual dependency.

$$\frac{\mathbf{p} : \mathbf{e}_1 \equiv \mathbf{e}_2 \in \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash \mathbf{e}_1 \equiv \mathbf{e}_2}\ [\equiv\text{-Reflect-Idea}]$$

In this version of extensional equality, we avoid dependence on the typing judgment by restricting extensionality to apply only when we have a proof that we can use in the current context. Unfortunately, this breaks subject reduction, and if equivalence is untyped, it can lead to unsoundness. Since the only purpose was to break the dependence between equivalence and typing, and subject reduction is important for a compiler IL, we end up with the standard reflection rule.

That said, these issues about typed equivalence are only about complicating the proof technique, not the translation. Extensional equivalence does not introduce any new dependencies that ANF can disrupt, so it seems likely that ANF would still be type preserving, but it will almost certainly be difficult to prove.

7.4 Recovering decidability

Our target language CC_e^A is extensional type theory, which is well known to have undecidable type checking. An IL with undecidable type checking affects our type-check-then-link approach, even though linking may not necessarily occur at CC_e^A. This is because the decidability of type checking of earlier ILs may affect the type checking of later ILs

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

in the compiler, where linking is to be done. If some information is lost resulting in undecidability, then that information is unlikely to be regained later.

There are then two main approaches to avoid undecidable type checking. One is to find ways to recover decidability for the subset of CC_e^A constructed by the compiler. The other is to remove equivalence reflection from CC_e^A and attempt to recover the proof of type preservation, which is much more difficult and not ideal for compilation.

Recovering Decidability for CC_e^A. Intuitively, the target terms in CC_e^A constructed by the compiler should be decidable. This is because the terms are constructed from terms in a decidable source language. In principle, the translation could make use of this fact and insert whatever annotations are necessary into the target term to ensure it can be decidably checked in an extensional type theory. Determining small, suitable annotations for compilation is the main subject of future work.

One approach for annotating the compiled term is a technique from proof-carrying code (PCC). In a variant of PCC by Necula & Rahul (2001), the inference rules are represented as a higher-order logic program, and the proof checker is a non-deterministic logic interpreter. The proofs are made deterministic by recording non-deterministic choices as a bit-string. Our compiler could be modified to produce a similar bit-string encoding the non-deterministic choices. The type checker could then be modified to interpret the encoding to ensure type checking is decidable.

If all else fails, we can recover decidability to translating typing derivations rather than syntax. Donning our propositions-as-types hat once again, we can obtain the desired typing derivation by using the proof of type preservation. The proof can be viewed as a function from source typing derivations to target typing derivations. The target typing derivation can then be translated to CIC with additional axioms and be decidably checked (Oury, 2005). However, this would require shipping the type preservation proof with the compiler, which might be undesirable.

Using a Target Language Without Equivalence Reflection. We could use the *convoy pattern* (Chlipala, 2013) to compile if expressions, as we did to show consistency of CC_e^A in Section 4.2.1. However, there are two issues with this approach: (1) the convoy pattern is *not* in ANF and (2) the resulting code generated just to show types are preserved (i.e., code not related to execution) is extremely large.

One could attempt to reduce the code size generated by the convoy pattern by using type safe coercions. Consider the addition of the following coercion, $\mathbf{subst}_{\,\mathbf{p}}\;\mathbf{e}$:

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

Naive additions of such coercions, however, result in a non-type-preserving translation. For example, the coercions could be inserted into the translation for **if** as follows:

$$\begin{aligned}
[\![\mathsf{if}\;\mathsf{e}\;\mathsf{then}\;\mathsf{e}_1\;\mathsf{else}\;\mathsf{e}_2]\!]\;\mathbf{K} \;\stackrel{\text{def}}{=}\;& [\![\mathsf{e}]\!]\;\mathbf{let}\;\mathbf{x} = [\cdot]\;\mathbf{in}\\
&\quad \mathbf{if}\;\mathbf{x}\;\mathbf{then}\\
&\quad ([\![\mathsf{e}_1]\!]\;\mathbf{let}\;\mathbf{y} = [\cdot]\;\mathbf{in}\;\mathbf{K}[\mathbf{subst}_{\,\mathbf{refl}\;\mathbf{x}}\;\mathbf{y}])\;\mathbf{else}\\
&\quad ([\![\mathsf{e}_2]\!]\;\mathbf{let}\;\mathbf{y} = [\cdot]\;\mathbf{in}\;\mathbf{K}[\mathbf{subst}_{\,\mathbf{refl}\;\mathbf{x}}\;\mathbf{y}])
\end{aligned}$$

The type preservation proof fails in the branches of the **if**, despite the coercion. This is because our judgmental equivalence cannot determine that the *expression* expected by the continuation, $[\![\text{if e then } e_1 \text{ else } e_2]\!]$, is equivalent to the expression $\mathbf{subst}_{\mathbf{refl\ x}}\ \mathbf{y}$. Equality reflection was key in proving this equivalence in our proof of type preservation.

7.5 CPS translation

ANF is favored as a compiler intermediate representation, although not universally. Cong *et al.* (2019) include a thorough summary of the arguments for and against both CPS and ANF as intermediate representations.

Most recent work on CPS translation of dependently typed languages focuses on expressing control effects (Miquey, 2017; Pédrot, 2017; Cong & Asai, 2018*a*, 2018b), which we discuss further in Section 7.7. When expressing control effects with dependent types, it is *necessary* for consistency to prevent certain dependencies from being expressed (Barthe & Uustalu, 2002; Herbelin, 2005), so these translations do not try to recover dependencies in the way we discuss in Section 2.

Two CPS translations exist that do try to recover dependencies in the absence of effects, but fail to scale to a predicative universe hierarchy. Bowman *et al.* (2018) present a type-preserving CPS translation for the Calculus of Constructions. They add a special form and typing rule for the application of a computation to a continuation which essentially records a machine step and is essentially similar to the **let** typing rule. They also add a non-standard equality rule that essentially corresponds to Lemma 4.15 (Naturality). Cong & Asai (2018*a*) extend the translation of Bowman *et al.* (2018) to dependent pattern matching using essentially the same typing rule for **if** as we do in CC_e^A.

Unfortunately, this rule relies on interpreting all functions as parametric and in a single computationally relevant impredicative universe. Both of these restrictions are a problem; Boulier *et al.* (2017) give an example of a type theory that is anti-parametric because it enables ad hoc polymorphism and type quotation. It is simpler to demonstrate the problems that occur when relying on impredicativity.

Impredicativity is inconsistent with large elimination, that is, eliminating a type to a term (Huet, 1986; Chan, 2021)[9] This means the CPS translation cannot be used for a consistent type theory with large elimination, while our ANF translation can. For a concrete example of what kind of expressiveness is regained, consider the following definition of the type of arbitrary arity functions.

Formally	Informally
$\text{nargs} = \lambda\, x : \text{Nat}.\ (\text{indnat Type}_0\ x$ $\quad \text{Nat}$ $\quad \lambda\, x : \text{Nat}.\ \lambda\, r : \text{Type}_0.\ \text{Nat} \to r)$	$\textit{nargs}\ 0 \quad = \quad \text{Nat}$ $\textit{nargs}\ \text{succ}\ x \quad = \quad \text{Nat} \to (\textit{nargs}\ x)$

This definition lets us build well-typed multi-arity functions over natural numbers. For example, the type of the n-ary summation function is $\textit{sum} : \Pi\, n : \text{Nat}.\ \textit{nargs}\ n$. But, this requires large elimination and would be inconsistent in an impredicative universe and thus cannot be supported by the CPS translation of Bowman *et al.* (2018). The desire for this

[9] The idea for this proof is due to Huet (1986); however, he seems to have left the proof as an exercise to the reader. The idea seems quite well understood in the type theory folklore, but Chan (2021) formalizes the full proof in Agda.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

sort of definitions makes a predicative universe hierarchy almost, but not quite, universal in dependently typed languages. Cedille is a notable exception, instead trying to take full advantage of impredicativity (Stump & Jenkins, 2018).

7.6 Call-by-push-value and monadic form

Call-by-push-value (CBPV) is similar to our ANF target language and to CPS target languages. In essence, CBPV is a λ calculus in monadic form suitable for reasoning about call-by-value (CBV) or call-by-name (CBN), due to explicit sequencing of computations (Levy, 2012). It has values, computations, and continuations, as we do, and has compositional typing rules (which inspired much of our own presentation). The structure of CBPV is of useful for modeling effects; all computations should be considered to carry an arbitrary effect, while values do not.

Work on designing a dependent call-by-push-value (dCBPV) runs into some of the same design issues that we see in ANF (Ahman, 2017; Vákár, 2017; Pédrot & Tabareau, 2019), but critically avoids the central difficulties introduced in Section 2. The reason is essentially that monadic form is more compositional than ANF, so dependency is not disrupted in the same way.

Recall from Section 4 that our definition of composition was entirely motivated by the need to compose configurations and continuations. In monadic form generally, there is no distinction between computation and configurations, and `let` is free to compose configurations. This means that configurations can return intermediate computations, instead of composing the entire rest of the continuation inside the body of a `let`. The monadic translation of $\mathsf{snd}\ \mathsf{e}$, which is problematic in ANF, is given below and is easily type preserving.

$$[\![\mathsf{snd}\ \mathsf{e} : \mathsf{B}[\mathsf{y} := \mathsf{e}]]\!] = \mathbf{let}\ \mathbf{x} = [\![\mathsf{e}]\!]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x} : [\![\mathsf{B}]\!][\mathbf{y} := [\![\mathsf{e}]\!]]$$

Note that since `let` can bind the "configuration" $[\![\mathsf{e}]\!]$, the typing rule [LET] and the compositionality lemma suffice to show type preservation, without any reasoning about definitions. In fact, we do not even need *definitions* for monadic form; we only need a dependent result type for `let`. A similar argument applies to the monadic translation of dependent `if`: since we can still nest `if` on the right-hand side of a `let`, the difficulties we encounter in the ANF translation are avoided.

The dependent typing rule for `let` without definitions is essentially the rule given by Vákár (2017), called the dependent Kleisli extension, to support the CBV monadic translation of type theory into dCBPV, and the CBN translation with strong dependent pairs. Vákár (2017) observes that without the dependent Kleisli extension, CBV translation is ill-defined (not type preserving), and CBN only works for dependent elimination of positive types. This is the same as the observation made independently by Bowman *et al.* (2018) that type-preserving CBV CPS fails for Π types, in addition to the well-known result that the CBN translation failed for Σ types (Barthe & Uustalu, 2002).

Recently, Pédrot & Tabareau (2019) introduce another variant of dependent call-by-push-value dubbed δCBPV and discuss the thunkability extension in order to develop a monadic translation to embed from CC_ω into δCBPV. Thunkability expresses that a computation behaves like a pure computation, and so it can be depended upon. This justifies the addition of an equation to their type theory that is similar to our Lemma 4.15, but should hold even when the language is extended with effects. The authors note that the

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

thunkability extensions seems to require that the target of the thunkable translation be an extensional type theory.

Monadic form has been studied for compilation (Benton *et al*., 1998; Benton & Kennedy, 1999). In these instances, the compiler simplifies from monadic form to ANF as a series of intra-language rewrites. Once in ANF, that is, a form that is normal with respect to the commuting conversions within monadic form, code generation is greatly simplified since all nesting has been eliminated. This more practical implementation technique, not strictly adhering to ANF until necessary, is also used in GHC (Maurer *et al*., 2017) and Chez Scheme.[10]

These commutative conversions are called commutative cuts in the dependent type theory literature (Herbelin, 2009; Boutillier, 2012). Formally, the problem of commutative cuts can be phrased as: Is the following transformation type preserving?

$$\mathsf{K}[\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2] \rightsquigarrow \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{K}[\mathsf{e}_1]\ \mathsf{else}\ \mathsf{K}[\mathsf{e}_2]$$

ANF necessarily performs this transformation, as we have shown.

These same commuting conversion *do not* preserve typing in a standard dependently typed language, but *do* in our target language CC_e^A. While we study this through the ANF translation directly, the more practical application of this work is likely in the design of the target language, which is designed to support ANF, rather than in the ANF translation itself. In this view, the ANF translation can be seen as a proof technique that all intermediate rewrites from monadic form to ANF are support in the target language.

7.7 CPS for control effects in dependent types

Most work on CPS for dependent types is primarily concerned with integrating control effects and dependent types, not with type-preserving compilation. Therefore, these CPS translations necessarily make different design choices than we describe in Section 2. For example, we want to avoid any restriction on the source language. However, one must necessarily restrict certain dependencies to integrate control effects and dependent types, at least for the effectful parts of the language. For our translation, naturality is important, but it is non-goal for effectful terms, or is a goal only up to a boundary. We briefly describe some of these translations.

To justify soundness of a dependent sequent calculus, Miquey (2017) present a CPS translation based on double negation. Sequent calculi make co-values (essentially, continuation) explicit in the language and thus express control effects. Miquey (2017) develop their CPS translation and avoid relying on parametricity and impredicativity. They rely on modified source type system to track equalities when checking co-values that depend on expression, which is essentially similar to the definitions in our continuation typing. They use delimited continuations to enforce naturality at certain boundaries, to constrain the scope of effects. They also rely on the *negative-elimination-free* (NEF) restriction (Herbelin, 2012), which disallows proofs that contain certain kinds of computations, such as arbitrary dependent projection from Σ types. If we want to compile existing languages, such as Coq, which have significant code bases, we cannot admit restrictions like the NEF restriction.

[10] `https://github.com/cisco/ChezScheme`.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

7.8 Non-dependently typed ANF translation

There has not been much work on type preservation for simply typed (or, more specifically, non-dependently typed) ANF translation. Most of the type preservation literature focuses on CPS translation (Shao, 1997; Morrisett *et al*., 1999; Thielecke, 2003; Kennedy, 2007; Ahmed & Blume, 2011; Perconti & Ahmed, 2014; Patterson *et al*., 2017).

This is probably because ANF translation without dependent types is, essentially, not interesting. CPS translation transforms every expression, making explicit a representation of computation and continuation. This requires a (typed) encoding that captures all the properties of interest and changes the types of every term change. This makes type preservation non-trivial and raises interesting questions about exactly type computations and continuations should be assigned. This can be particularly interesting when considering a compiled component interacting with a hostile context (Ahmed & Blume, 2011).

By contrast, ANF essentially just reorders existing terms and binds them to names without introducing any new objects into the syntax. No types change at all, so long as types do not depend on the syntax of terms.

A cursory study of non-dependently typed ANF appears in the technical appendix by Ahmed & Blume (2011), companion to Ahmed & Blume (2011), which briefly studies ANF translation of System F. They show that, in System F, ANF is type preserving and fully abstract, since System F in ANF is actually just a subset of System F. The proof is not given in detail, however.

Another tangential study of type-preserving ANF appears in the work of Saffrich & Thiemann (2020), which use ANF as part of a proof to show that imperative session types are subsumed by a functional session types API. The correctness and type-preservation of ANF translation is similarly considered trivial in the text since ANF does not affect typing, although they provide a detailed proof of type preservation in the appendix.

8 Conclusion

We develop a type-preserving ANF translation for ECC—a significant subset of Coq including dependent functions, dependent pairs, dependent elimination of booleans, natural numbers, and the infinite hierarchy of universes—and prove correctness of separate compilation with respect to a machine semantics for the target language. The translation, these proofs, and our proof technique for ANF are main contributions. This translation provides strong evidence that type-preserving compilation can support all of dependent type theory. It gives insights into type preservation for dependent types in general and into related work on type-preserving control flow transformations.

Acknowledgments

We gratefully acknowledge Youyou Cong, Max S. New, Hugo Herbelin, Matthias Felleisen, Greg Morrisett, Simon Peyton Jones, Paul Downen, Andrew Kennedy, Brian LaChance, Danel Ahman, Carlo Angiuli, and the many anonymous reviewers for their time in discussing ANF, CPS and related problems during the course of this work. Thank you all. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207. Cette recherche a

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207.

Conflicts of interest

None.

References

Ahman, D. (2017) *Fibred Computational Effects*. Ph.D. thesis. University of Edinburgh. Available at: http://arxiv.org/abs/1710.02594.

Ahmed, A. (2015) Verified Compilers for a Multi-language World. In Summit on Advances in Programming Languages (SNAPL). 10.4230/LIPIcs.SNAPL.2015.15.

Ahmed, A. & Blume, M. (2011) An equivalence-preserving CPS translation via multi-language semantics. In International Conference on Functional Programming (ICFP). 10.1145/2034773.2034830.

Ahmed, A. & Blume, M. (2011) *An Equivalence-Preserving CPS Translation via Multi-Language Semantics (Technical Appendix)*. Technical report. Available at: http://www.ccs.neu.edu/home/amal/papers/epc-tr.pdf.

Anand, A., Appel, A. W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O. S., Sozeau, M. & Weaver, M. (2017) CertiCoq: A verified compiler for Coq. In International Workshop on Coq for Programming Languages (CoqPL). Available at: http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf.

Appel, A. W. (2015) Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst. (TOPLAS)*. **37**(2). 10.1145/2701415.

Barthe, G., Grégoire, B. & Zanella-Béguelin, S. (2009) Formal certification of code-based cryptographic proofs. In Symposium on Principles of Programming Languages (POPL). 10.1145/1480881.1480894.

Barthe, G., Hatcliff, J. & Sørensen, M. H. B. (1999) CPS translations and applications: The cube and beyond. *Higher-Order Symb. Comput*. **12**(2). 10.1023/a:1010000206149.

Barthe, G. & Uustalu, T. (2002) CPS translating inductive and coinductive types. In Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM). 10.1145/509799.503043.

Benton, N. & Kennedy, A. (1999) Monads, effects and transformations. *Electron. Notes Theoret. Comput. Sci*. **26**, 3–20. 10.1016/s1571-0661(05)80280-4.

Benton, N., Kennedy, A. & Russell, G. (1998) Compiling standard ML to Java bytecodes. In International Conference on Functional Programming (ICFP). 10.1145/289423.289435.

Boulier, S., Pédrot, P. & Tabareau, N. (2017) The next 700 syntactical models of type theory. In Conference on Certified Programs and Proofs (CPP). 10.1145/3018610.3018620.

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

Bowman, W. J. & Ahmed, A. (2018) Parametric Closure Conversion for CIC. Available at: https://web.archive.org/web/20210423031005/https://www.williamjbowman.com/resources/wjb2018-techreport-parametric-cc-cic.pdf.

Bowman, W. J., Cong, Y., Rioux, N. & Ahmed, A. (2018) Type-preserving CPS translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang. (PACMPL)*. **2**(POPL). 10.1145/3158110.

Briski, K. A., Chitale, P., Hamilton, V., Pratt, A., Starr, B., Veroulis, J. & Villard, B. (2008) Minimizing Code Defects to Improve Software Quality and Lower Development Costs. *Development Solutions. IBM.* Available at: ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf.

Chan, J. (2021) An Analysis of An Analysis of Girard's Paradox. Available at: https://web.archive.org/web/20220429152422/https://ionathan.ch//2021/11/24/inconsistencies.html.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Chan, J. & Bowman, W. J. (2020) Practical sized typing for coq. Available at: https://arxiv.org/abs/1912.05601.

Chen, J., Hawblitzel, C., Perry, F., Emmi, M., Condit, J., Coetzee, D. & Pratikaki, P. (2008) Type-preserving compilation for large-scale optimizing object-oriented compilers. In International Conference on Programming Language Design and Implementation (PLDI). 10.1145/1375581.1375604.

Chlipala, A. (2007) A certified type-preserving compiler from lambda calculus to assembly language. In International Conference on Programming Language Design and Implementation (PLDI). 10.1145/1250734.1250742.

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

Cong, Y. & Asai, K. (2018a) Handling delimited continuations with dependent types. *Proc. ACM Program. Lang. (PACMPL)*. **2**(ICFP). 10.1145/3236764.

Cong, Y. & Asai, K. (2018b) Shifting and resetting in the calculus of constructions. In International Symposium on Trends in Functional Programming (TFP). Available at: https://sites.google.com/site/youyoucong212/tfp-2018.

Cong, Y., Osvald, L., Essertel, G. M. & Rompf, T. (2019) Compiling with continuations, or without? whatever. *PACMPL* **3**(ICFP), 79:1–79:28. 10.1145/3341643.

Coquand, T. & Huet, G. (1988) The calculus of constructions. *Inf. Comput*. **76**(2–3). 10.1016/0890-5401(88)90005-3.

Flanagan, C., Sabry, A., Duba, B. F. & Felleisen, M. (1993) The essence of compiling with continuations. In International Conference on Programming Language Design and Implementation (PLDI). 10.1145/155090.155113.

Gordon, A. D. (1995) Bisimilarity as a theory of functional programming. In Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, LA, USA, March 29–April 1, 1995. Elsevier, pp. 232–252. 10.1016/S1571-0661(04)80013-6.

Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X. n., Weng, S.-c., Zhang, H. & Guo, Y. (2015) Deep specifications and certified abstraction layers. In Symposium on Principles of Programming Languages (POPL). 10.1145/2775051.2676975.

Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V. & Costanzo, D. (2016) CertiKOS: An extensible architecture for building certified concurrent OS kernels. In Symposium on Operating Systems Design and Implementation (OSDI). Available at: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

Herbelin, H. (2005) On the degeneracy of Σ-types in presence of computational classical logic. In International Conference on Typed Lambda Calculi and Applications. 10.1007/11417170_16.

Herbelin, H. (2009) On a few open problems of the calculus of inductive constructions and on their practical consequences. Updated 2010. Available at: https://web.archive.org/web/20181125182737/http://pauillac.inria.fr/~herbelin/talks/cic.pdf.

Herbelin, H. (2012) A constructive proof of dependent choice, compatible with classical logic. In Symposium on Logic in Computer Science (LICS). 10.1109/lics.2012.47.

Huet, G. (1986) *Formal Structures for Computation and Deduction*. Technical report. Available at: http://web.archive.org/web/20220408113829/http://pauillac.inria.fr/~huet/PUBLIC/Formal_Structures.ps.gz.

Kennedy, A. (2007) Compiling with continuations, continued. In International Conference on Functional Programming (ICFP). 10.1145/1291220.1291179.

Lennon-Bertrand, M., Maillard, K., Tabareau, N. & Tanter, E. (2022) Gradualizing the calculus of inductive constructions. *ACM Trans. Program. Lang. Syst*. 10.1145/3495528.

Leroy, X. (2009) A formally verified compiler back-end. *J. Autom. Reas*. **43**(4). 10.1007/s10817-009-9155-4.

Levy, P. B. (2012) *Call-By-Push-Value*. Available at: https://www.worldcat.org/oclc/7330961929.

Luo, Z. (1990) *An Extended Calculus of Constructions*. Ph.D. thesis. University of Edinburgh. Available at: http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-118/.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Maurer, L., Downen, P., Ariola, Z. M. & Peyton Jones, S. (2017) Compiling without continuations. In International Conference on Programming Language Design and Implementation (PLDI). 10.1145/3062341.3062380.

Miquey, É. (2017) A classical sequent calculus with dependent types. In European Symposium on Programming (ESOP). 10.1007/978-3-662-54434-1_29.

Morrisett, G., Walker, D., Crary, K. & Glew, N. (1999) From system F to typed assembly language. *ACM Trans. Program. Lang. Syst. (TOPLAS)*. **21**(3). 10.1145/319301.319345.

Necula, G. C. (1997) Proof-carrying code. In Symposium on Principles of Programming Languages (POPL). 10.1145/263699.263712.

Necula, G. C. & Rahul, S. P. (2001) Oracle-based checking of untrusted software. In Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17–19, 2001. ACM, pp. 142–154. 10.1145/360204.360216.

Oury, N. (2005) Extensionality in the calculus of constructions. In Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22–25, 2005, Proceedings. Springer, pp. 278–293. 10.1007/11541868_18.

Patterson, D., Perconti, J., Dimoulas, C. & Ahmed, A. (2017) FunTAL: Reasonably mixing a functional language with assembly. In International Conference on Programming Language Design and Implementation (PLDI). 10.1145/3062341.3062347.

Pédrot, P. (2017) A parametric CPS to sprinkle CIC with classical reasoning. In Workshop on Syntax and Semantics of Low-Level Languages. Available at: https://web.archive.org/web/20220122222238/https://www.cs.bham.ac.uk/~zeilbern/lola2017/abstracts/LOLA_2017_paper_5.pdf.

Pédrot, P.-M. & Tabareau, N. (2019) The fire triangle: How to mix substitution, dependent elimination, and effects. In Symposium on Principles of Programming Languages (POPL). 10.1145/3371126.

Perconti, J. T. & Ahmed, A. (2014) Verifying an open compiler using multi-language semantics. In European Symposium on Programming (ESOP). 10.1007/978-3-642-54833-8_8.

Peyton Jones, S. L. (1996) Compiling Haskell by program transformation: A report from the trenches. In European Symposium on Programming (ESOP). 10.1007/3-540-61055-3_27.

Pitts, A. M. (1997) Operationally-based theories of program equivalence. In *Semantics and Logics of Computation*, Dybjer, P. & Pitts, A. M. (eds), Publications of the Newton Institute. Cambridge University Press, pp. 241–298. 10.1017/CBO9780511526619.007.

Sabry, A. & Felleisen, M. (1992) Reasoning about programs in continuation-Passing style. In LISP and Functional Programming (LFP). 10.1145/141478.141563.

Sabry, A. & Wadler, P. (1997) A reflection on call-by-value. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **19**(6). 10.1145/267959.269968.

Saffrich, H. & Thiemann, P. (2020) *Relating functional and imperative session types*. Available at: https://arxiv.org/abs/2010.08261.

Sarkar, S., Pientka, B. & Crary, K. (2005) Small proof witnesses for LF. In International Conference Logic Programming (ICLP). 10.1007/11562931_29.

Severi, P. & Poll, E. (1994) Pure type systems with definitions. In International Symposium Logical Foundations of Computer Science (LFCS). 10.1007/3-540-58140-5_30.

Shao, Z. (1997) *An Overview of the FLINT/ML Compiler*. Available at: https://web.archive.org/web/20161125002746/http://cs.bc.edu/~muller/TIC97//Shao.ps.gz.

Shao, Z., League, C. & Monnier, S. (1998) Implementing typed intermediate languages. In International Conference on Functional Programming (ICFP). 10.1145/289423.289460.

Shao, Z., Trifonov, V., Saha, B. & Papaspyrou, N. (2005) A type system for certified binaries. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **27**(1). 10.1145/1053468.1053469.

Sozeau, M. (2008) *Un environnement pour la programmation avec types dépendants. (An Environment for Programming with Dependent Types)*. Ph.D. thesis. Orsay, France: University of Paris-Sud. Available at: https://tel.archives-ouvertes.fr/tel-00640052.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Stecklein, J. M., Dabney, J., Dick, B., Haskins, B., Lovell, R. & Moroney, G. (2004) *Error Cost Escalation through the Project Life Cycle*. Technical report. NASA. Available at: https://ntrs.nasa.gov/search.jsp?R=20100036670.
Stewart, G., Beringer, L., Cuellar, S. & Appel, A. W. (2015) Compositional CompCert. In Symposium on Principles of Programming Languages (POPL). 10.1145/2676726.2676985.
Stump, A. & Jenkins, C. (2018) *Syntax and Semantics of Cedille*. Available at: http://arxiv.org/pdf/1806.04709v3.
Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. & Lee, P. (1996) TIL: A type-directed optimizing compiler for ML. In International Conference on Programming Language Design and Implementation (PLDI). 10.1145/231379.231414.
Thielecke, H. (2003) From control effects to typed continuation passing. In Symposium on Principles of Programming Languages (POPL). 10.1145/640128.604144.
Timany, A. & Sozeau, M. (2017) Consistency of the predicative calculus of cumulative inductive constructions (pCuIC). arXiv preprint arXiv:1710.03912. Available at: https://arxiv.org/abs/1710.03912.
Vákár, M. (2017) *In Search of Effectful Dependent Types*. Ph.D. thesis. Oxford University. Available at: http://arxiv.org/abs/1706.07997.
Watkins, K., Cervesato, I., Pfenning, F. & Walker, D. (2003) A concurrent logical framework: The propositional fragment. In International Workshop on Types for Proofs and Programs (TYPES). 10.1007/978-3-540-24849-1_23.
Xi, H. & Harper, R. (2001) A dependently typed assembly language. In International Conference on Functional Programming (ICFP). 10.1145/507635.507657.

1 Appendix: Full proof of type preservation and extended figures

Lemma 1.1.

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

Proof The proof is by induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash e : A$. Cases [Ax-Prop], [Lam], [App], [Snd], [ElimNat], and [If] are given, as they are representative.

Note that for all sub-expressions e' of type A', $[\![e']\!] : [\![A']\!]$ holds by instantiating the induction hypothesis with the empty continuation $[\cdot] : ([\![e']\!] : [\![A']\!]) \Rightarrow [\![A']\!]$. The general structure of each case is similar; we proceed by induction on a sub-expression and prove the new continuation $\mathbf{K}'$ is well typed by applications of [K-Bind]. Proving the body of $\mathbf{K}'$ is well typed requires using Lemma 4.3. This requires proving that $\mathbf{K}'$ is composed with a configuration $\mathbf{M}$ equivalent to $[\![e]\!]$, and showing their types are equivalent. To show $\mathbf{M}$ is equivalent to $[\![e]\!]$, we use the Lemma 5.4 and Lemma 4.15 to allow for composing configuations and continuations. Additionally, since several typing rules subsitute sub-expressions into the type system, we use Lemma 5.5 in these cases to show the types of $\mathbf{M}$ and $[\![e]\!]$ are equivalent. Each non-value proof case focuses on showing these two equivalences.

Case: [Ax-Prop]

We must show that $[\![\Gamma]\!], \mathbf{\Gamma}' \vdash \mathbf{K}[\mathbf{Prop}\,] : \mathbf{B}$. This follows from Lemma 4.2.

Case: [Lam]

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Must show $[\![\Gamma]\!] \vdash \mathbf{K}[\boldsymbol{\lambda}\, \mathbf{x} : [\![A']\!].\, [\![e']\!]] : \mathbf{B}$. This follows from Lemma 4.2 if we can show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \boldsymbol{\lambda}\, \mathbf{x} : [\![A']\!].\, [\![e']\!] : \boldsymbol{\Pi}\, \mathbf{x} : [\![A']\!].\, [\![B']\!]$. This follows from [LAM] if we can show $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x} : [\![A']\!] \vdash [\![e']\!] : [\![B']\!]$. This follows by induction on the judgment $\Gamma, x : A' \vdash e' : B'$ with the empty continuation $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x} : [\![A']\!] \vdash [\cdot] : ([\![e']\!] : [\![B']\!]) \Rightarrow [\![B']\!]$.

Case: [APP]

Must show that $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e_1]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x_1}\ \mathbf{x_2}])) : \mathbf{B}$.

Let $\mathbf{K_1} = (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x_1}\ \mathbf{x_2}]))$. Our conclusion follows by induction on $\Gamma \vdash e_1 : \Pi\, x : A'.\, B'$ if we show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathbf{K_1} : ([\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!]) \Rightarrow \mathbf{B}$. By [K-BIND], we must show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!]$ and $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!] \vdash [\![e_2]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x_1}\ \mathbf{x_2}]) : \mathbf{B}$.

The first goal follows from induction on $\Gamma \vdash e_1 : \Pi\, x : A'.\, B'$ with the empty continuation $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\cdot] : ([\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!]) \Rightarrow [\![\Pi\, x : A'.\, B']\!]$.

The second goal also follows by induction on $\Gamma \vdash e_2 : A'$, but we must show $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!] \vdash \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x_1}\ \mathbf{x_2}] : ([\![e_2]\!] : [\![A']\!]) \Rightarrow \mathbf{B}$. By [K-BIND] again, we must show $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!] \vdash [\![e_2]\!] : [\![A']\!]$ (which again follows by induction with the empty continuation) and $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash \mathbf{K}[\mathbf{x_1}\ \mathbf{x_2}] : \mathbf{B}$.

This follows from Lemma 4.3 if we can show (1) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash \mathbf{x_1}\ \ \mathbf{x_2} : [\![B'[x := e_2]]\!]$ (2) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash [\![e_1\ e_2]\!] : [\![B'[x := e_2]]\!]$ and (3) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash [\![e_1\ e_2]\!] \equiv \mathbf{x_1}\ \mathbf{x_2}$.

By $\rhd_\delta$ and [$\equiv$-STEP], goal (1) changes to $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash [\![e_1]\!]\ \ [\![e_2]\!] : [\![B'[x := e_2]]\!]$. We have previously shown that $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e_1]\!] : \boldsymbol{\Pi}\, \mathbf{x} : [\![A']\!].\, [\![B']\!]$ and $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!] \vdash [\![e_2]\!] : [\![A']\!]$. Using these facts with [APP], we derive $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash [\![e_1]\!]\ \ [\![e_2]\!] : [\![B']\!][\mathbf{x} := [\![e_2]\!]]$. We have that $[\![B']\!][\mathbf{x} := [\![e_2]\!]] \equiv [\![B'[x := e_2]]\!]$ by Lemma 5.5 and derive our conclusion by [CONV].

By $\rhd_\delta$ and [$\equiv$-STEP], goal (3) changes to $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \stackrel{\delta}{=} [\![e_1]\!] : [\![\Pi\, x : A'.\, B']\!], \mathbf{x_2} \stackrel{\delta}{=} [\![e_2]\!] : [\![A']\!] \vdash [\![e_1\ e_2]\!] \equiv [\![e_1]\!]\ \ [\![e_2]\!]$. We find that the left-hand side of the equivalence can be converted as well:

$$
\begin{aligned}
& [\![e_1\ e_2]\!] \\
& \stackrel{\text{def}}{=} [\![e_1]\!]\ \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{x_1}\ \mathbf{x_2} \\
& = \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{x_1}\ \mathbf{x_2} \langle\!\langle [\![e_1]\!] \rangle\!\rangle && \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ x_1 = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ x_1\ \mathbf{x_2} && \text{by Lemma 4.15} \\
& \equiv_\zeta [\![e_2]\!]\ \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ [\![e_1]\!]\ \ \mathbf{x_2} \\
& = \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ [\![e_1]\!]\ \ \mathbf{x_2} \langle\!\langle [\![e_2]\!] \rangle\!\rangle && \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ x_2 = [\![e_2]\!]\ \mathsf{in}\ [\![e_1]\!]\ \ x_2 && \text{by Lemma 4.15} \\
& \equiv_\zeta [\![e_1]\!]\ \ [\![e_2]\!]
\end{aligned}
$$

Then goal (2) follows from combining the fact that $[\![e_1\ e_2]\!] \equiv [\![e_1]\!]\ \ [\![e_2]\!]$ and $[\![e_1]\!]\ \ [\![e_2]\!] : [\![B'[x := e_2]]\!]$.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Case: [SND]

Must show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e]\!]\,(\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x_1}]) : \mathbf{B}$. This follows by the induction hypothesis for the sub-derivation $\Gamma \vdash e : \Sigma\, x : A'.\, B'$ if we can show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x_1}] : ([\![e]\!] : [\![\Sigma\, x : A'.\, B']\!]) \Rightarrow \mathbf{B}$.

By [K-BIND], it suffices to show (1) $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!]$ and (2) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{K}[\mathbf{snd}\ \mathbf{x_1}] : \mathbf{B}$.

Goal (1) follows by the induction hypothesis for $\Gamma \vdash e : \Sigma\, x : A'.\, B'$ with the well-typed empty continuation $[\cdot] : ([\![e]\!] : [\![\Sigma\, x : A'.\, B']\!]) \Rightarrow [\![\Sigma\, x : A'.\, B']\!]$.

Goal (2) follows by Lemma 4.3 if we can show (3) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{snd}\ \mathbf{x_1} : [\![B'[x := \mathsf{fst}\ e]]\!]$, (4) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash [\![\mathsf{snd}\ e]\!] \equiv \mathbf{snd}\ \mathbf{x_1}$, and (5) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash [\![\mathsf{snd}\ e]\!] : [\![B'[x := \mathsf{fst}\ e]]\!]$.

By $\rhd_\delta$, we are trying to show $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{snd}\ [\![e]\!] : [\![B'[x := \mathsf{fst}\ e]]\!]$. We have previously shown that $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e]\!] : \mathbf{\Sigma}\ \mathbf{x} : [\![A']\!].\, [\![B']\!]$. Using this fact with [SND], we derive that $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{snd}\ [\![e]\!] : [\![B']\!][\mathbf{x} := \mathbf{fst}\ [\![e]\!]]$. We can derive our goal by [CONV] if we can show that $[\![B'[x := \mathsf{fst}\ e]]\!]$ is equivalent to $[\![B']\!][\mathbf{x} := \mathbf{fst}\ [\![e]\!]]$. By Lemma 5.5, we have that $[\![B'[x := \mathsf{fst}\ e]]\!]$ is equivalent to $[\![B']\!][\mathbf{x} := [\![\mathsf{fst}\ e]\!]]$. Focusing on $[\![\mathsf{fst}\ e]\!]$, we have:

$$
\begin{aligned}
& [\![\mathsf{fst}\ e]\!] \\
& \overset{\text{def}}{=} [\![e]\!]\,(\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x_1}) \\
& = \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x_1} \langle\!\langle\, [\![e]\!]\, \rangle\!\rangle && \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ x_1 = [\![e]\!]\ \ \mathsf{in}\ \mathbf{fst}\ x_1 && \text{by Lemma 4.15} \\
& \rhd_\zeta\ \mathbf{fst}\ \ [\![e]\!]
\end{aligned}
$$

Finally, $[\![B'[x := \mathsf{fst}\ e]]\!]$ is equivalent to $[\![B']\!][\mathbf{x} := \mathbf{fst}\ [\![e]\!]]$ by [$\equiv$-SUBST].

By $\rhd_\delta$ and [$\equiv$-STEP], we are trying to show $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} [\![e]\!] : [\![\Sigma\, x : A'.\, B']\!] \vdash [\![\mathsf{snd}\ e]\!] \equiv \mathbf{snd}\ \ [\![e]\!]$. Focusing on $[\![\mathsf{snd}\ e]\!]$, we have:

$$
\begin{aligned}
& [\![\mathsf{snd}\ e]\!] \\
& \overset{\text{def}}{=} [\![e]\!]\,(\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x_1}) \\
& = \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x_1} \langle\!\langle\, [\![e]\!]\, \rangle\!\rangle && \text{by Lemma 5.4} \\
& \equiv \mathsf{let}\ x_1 = [\![e]\!]\ \ \mathsf{in}\ \mathbf{snd}\ x_1 && \text{by Lemma 4.15} \\
& \rhd_\zeta\ \mathbf{snd}\ \ [\![e]\!]
\end{aligned}
$$

Combining goals (3) and (4), we can show goal (5).

Case: [IF]

Must show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e]\!]\,(\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \ [\![e_1]\!]\ \mathbf{K}\ \mathbf{else}\ \ [\![e_2]\!]\ \mathbf{K}) : \mathbf{B}$.

Let $\mathbf{K_1} = \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \ [\![e_1]\!]\ \mathbf{K}\ \mathbf{else}\ \ [\![e_2]\!]\ \mathbf{K}$. Our conclusion follows by induction on $\Gamma \vdash e : \mathsf{Bool}$ if we show $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathbf{K_1} : ([\![e]\!] : [\![\mathsf{Bool}]\!]) \Rightarrow \mathbf{B}$. By [K-BIND], we must show (1) $[\![\Gamma]\!], \mathbf{\Gamma'} \vdash [\![e]\!] : [\![\mathsf{Bool}]\!]$ and (2) $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x} \overset{\delta}{=} [\![e]\!] : [\![\mathsf{Bool}]\!] \vdash \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \ [\![e_1]\!]\ \mathbf{K}\ \mathbf{else}\ \ [\![e_2]\!]\ \mathbf{K} : \mathbf{B}$.

Goal (1) follows from induction on $\Gamma \vdash e : \mathsf{Bool}$ with the empty continuation $[\cdot] : ([\![e]\!] : [\![\mathsf{Bool}]\!]) \Rightarrow [\![\mathsf{Bool}]\!]$.

By [IF] in goal (2), we focus on showing the new sub-goal $[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x} \overset{\delta}{=} [\![e]\!] : [\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash [\![e_1]\!]\ \mathbf{K} : \mathbf{B}$ as it is the most interesting.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Universes	U	::=	$\mathsf{Prop} \mid \mathsf{Type}_i$
Expressions	$\mathsf{e}, \mathsf{A}, \mathsf{B}$	::=	$\mathsf{x} \mid \mathsf{U} \mid \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \mid \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e} \mid \mathsf{e}\ \mathsf{e}$
		$\mid$	$\Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle\ \mathsf{as}\ \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \mid \mathsf{fst}\ \mathsf{e}$
		$\mid$	$\mathsf{snd}\ \mathsf{e} \mid \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{Bool}$
		$\mid$	$\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$
		$\mid$	$\mathsf{Nat} \mid \mathsf{zero} \mid \mathsf{succ}\ \mathsf{e} \mid \mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2$
Environments	Γ	::=	$\cdot \mid \Gamma, \mathsf{x} : \mathsf{A} \mid \Gamma, \mathsf{x} \stackrel{\delta}{=} \mathsf{e} : \mathsf{A}$

Fig. A15: ECC syntax.

This follows by Lemma 4.3 if we can show (3) $[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \stackrel{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash [\![\mathsf{e}_1]\!] : [\![\mathsf{B}'[\mathsf{x}' := \mathsf{e}]]\!]$, (4) $[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \stackrel{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash [\![\mathsf{e}_1]\!] \equiv [\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]$ and (5) $[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \stackrel{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash [\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!] : [\![\mathsf{B}'[\mathsf{x}' := \mathsf{e}]]\!]$.
Goal (3) follows from induction with the empty continuation and [Conv] if we can show that $[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \stackrel{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash [\![\mathsf{B}'[\mathsf{x}' := \mathsf{e}]]\!] \equiv [\![\mathsf{B}'[\mathsf{x}' := \mathsf{true}]]\!]$. By Lemma 5.5, we are trying to show $[\![\mathsf{B}']\!][\mathbf{x}' := [\![\mathsf{e}]\!]] \equiv [\![\mathsf{B}']\!][\mathbf{x}' := \mathbf{true}]$. By [≡-Reflect] and [Var], as well as equivalence by $\triangleright_\delta$, we can derive that under the extended environment $[\![\mathsf{e}]\!] \equiv \mathbf{true}$, and we conclude with [≡-Subst].
Focusing the right-hand side of goal (4), we have:

$$
\begin{aligned}
&[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!] \\
&\stackrel{\text{def}}{=} [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!]) \\
&= \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!]\ \langle\!\langle [\![\mathsf{e}]\!] \rangle\!\rangle \\
&\text{by Lemma 5.4} \\
&\equiv \mathsf{let}\ \mathsf{x} = [\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{if}\ \mathsf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!] \\
&\text{by Lemma 4.15} \\
&\triangleright_\zeta\ \mathbf{if}\ [\![\mathsf{e}]\!]\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{else}\ [\![\mathsf{e}_2]\!] \\
&\equiv [\![\mathsf{e}_1]\!] \\
&\text{by [≡-If-}\beta_1\text{] since } [\![\mathsf{e}]\!] \equiv \mathbf{true} \text{ by [≡-Reflect] and [Var]}
\end{aligned}
$$

Combining goals (3) and (4), we can show goal (5).

Case: [ElimNat]

Must show

$[\![\Gamma]\!], \mathbf{\Gamma}' \vdash [\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{indnat}\ [\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2]))) : \mathbf{B}$.

This can be proven using the techniques in the [App] case: induction on subexpressions and showing that each continuation is well typed. The body of the final **let** expression must be shown to be of type **B**, which can be proven by Lemma 4.2 if we show

$[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{y} \stackrel{\delta}{=} [\![\mathsf{e}]\!] : [\![\mathsf{Nat}]\!], \mathbf{x}_1 \stackrel{\delta}{=} [\![\mathsf{e}_1]\!] : [\![\mathsf{A}[\mathsf{zero} := \mathsf{x}]]\!], \mathbf{x}_2 \stackrel{\delta}{=} [\![\mathsf{e}_2]\!] :$
$[\![\Pi\, \mathsf{n} : \mathsf{Nat}.\, \Pi\, \mathsf{x}' : \mathsf{A}[\mathsf{x} := \mathsf{n}].\, \mathsf{A}[\mathsf{x} := \mathsf{succ}\ \mathsf{n}]]\!] \vdash [\![\mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2]\!] \equiv \mathbf{indnat}\ [\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2$

This can be done by focusing on the left-hand side of the equivalence:

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$[\![\text{indnat A e } e_1 \; e_2]\!]$

$\stackrel{\text{def}}{=} [\![e]\!] \; (\textbf{let y} = [\cdot] \textbf{ in } [\![e_1]\!] \; (\textbf{let } x_1 = [\cdot] \textbf{ in } [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; \textbf{y } x_1 \; x_2)))$

$= \textbf{let y} = [\cdot] \textbf{ in } [\![e_1]\!] \; (\textbf{let } x_1 = [\cdot] \textbf{ in } [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; \textbf{y } x_1 \; x_2))\langle\!\langle [\![e]\!] \rangle\!\rangle$

by Lemma 5.4

$\equiv \text{let y} = [\![e]\!] \text{ in } [\![e_1]\!] \; (\textbf{let } x_1 = [\cdot] \textbf{ in } [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; \textbf{y } x_1 \; x_2))$

by Lemma 4.15

$\rhd_\zeta [\![e_1]\!] \; (\textbf{let } x_1 = [\cdot] \textbf{ in } [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; [\![e]\!] \; x_1 \; x_2))$

$= \textbf{let } x_1 = [\cdot] \textbf{ in } [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; [\![e]\!] \; x_1 \; x_2)\langle\!\langle [\![e_1]\!] \rangle\!\rangle$

by Lemma 5.4

$\equiv \text{let } x_1 = [\![e_1]\!] \text{ in } [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; [\![e]\!] \; x_1 \; x_2)$

by Lemma 4.15

$\rhd_\zeta [\![e_2]\!] \; (\textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; [\![e]\!] \; [\![e_1]\!] \; x_2)$

$= \textbf{let } x_2 = [\cdot] \textbf{ in indnat } [\![A]\!] \; [\![e]\!] \; [\![e_1]\!] \; x_2 \langle\!\langle [\![e_2]\!] \rangle\!\rangle$

by Lemma 5.4

$\equiv \text{let } x_2 = [\![e_2]\!] \text{ in } \textbf{indnat } [\![A]\!] \; [\![e]\!] \; [\![e_1]\!] \; x_2$

by Lemma 4.15

$\rhd_\zeta \textbf{ indnat } [\![A]\!] \; [\![e]\!] \; [\![e_1]\!] \; [\![e_2]\!]$ ∎

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

$$
\begin{array}{rcll}
x & \rhd_\delta & e & \text{where } x \stackrel{\delta}{=} e : A \in \Gamma \\
(\lambda\, x : A.\, e_1)\; e_2 & \rhd_\beta & e_1[x := e_2] & \\
\text{fst } \langle e_1, e_2 \rangle & \rhd_{\sigma_1} & e_1 & \\
\text{snd } \langle e_1, e_2 \rangle & \rhd_{\sigma_2} & e_2 & \\
\text{let } x = e_1 \text{ in } e_2 & \rhd_\zeta & e_2[x := e_1] & \\
\text{if true then } e_1 \text{ else } e_2 & \rhd_{\mathbb{B}_1} & e_1 & \\
\text{if false then } e_1 \text{ else } e_2 & \rhd_{\mathbb{B}_2} & e_2 & \\
\text{indnat A zero } e_1\; e_2 & \rhd_{\iota_1} & e_1 & \\
\text{indnat A (succ e) } e_1\; e_2 & \rhd_{\iota_2} & (e_2\; e)\; (\text{indnat A e } e_1\; e_2) &
\end{array}
$$

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

$$
\cdots \frac{\Gamma, x \stackrel{\delta}{=} e : A \vdash e_1 \rhd^* e_2}{\Gamma \vdash \text{let } x = e \text{ in } e_1 \rhd^* \text{let } x = e \text{ in } e_2}\;[\text{Red-Cong-Let}] \qquad \frac{}{\Gamma \vdash e \rhd^* e}\;[\text{Red-Refl}]
$$

$$
\frac{\Gamma \vdash e \rhd e_1 \qquad \Gamma \vdash e_1 \rhd^* e'}{\Gamma \vdash e \rhd^* e'}\;[\text{Red-Trans}]
$$

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

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

Fig. A16: ECC dynamic semantics (excerpt).

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$$\frac{\Gamma \vdash \mathsf{A} \rhd^* \mathsf{A}' \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} \rhd^* \mathsf{e}'}{\Gamma \vdash \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e} \rhd^* \lambda\, \mathsf{x} : \mathsf{A}'.\, \mathsf{e}'} \text{[Red-Cong-Lam]}$$

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

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

$$\frac{\Gamma \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_1' \qquad \Gamma \vdash \mathsf{e}_2 \rhd^* \mathsf{e}_2' \qquad \Gamma \vdash \mathsf{A} \rhd^* \mathsf{A}'}{\Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \text{ as } \mathsf{A} \rhd^* \langle \mathsf{e}_1', \mathsf{e}_2' \rangle \text{ as } \mathsf{A}'} \text{[Red-Cong-Pair]}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_1' \qquad \Gamma \vdash \mathsf{e}_2 \rhd^* \mathsf{e}_2'}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 \rhd^* \mathsf{e}_1'\ \mathsf{e}_2'} \text{[Red-Cong-App]} \qquad \frac{\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'}{\Gamma \vdash \mathsf{fst}\ \mathsf{e} \rhd^* \mathsf{fst}\ \mathsf{e}'} \text{[Red-Cong-Fst]}$$

$$\frac{\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'}{\Gamma \vdash \mathsf{snd}\ \mathsf{e} \rhd^* \mathsf{snd}\ \mathsf{e}'} \text{[Red-Cong-Snd]}$$

$$\frac{\Gamma, \mathsf{x} \stackrel{\delta}{=} \mathsf{e} : \mathsf{A} \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_2}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}_1 \rhd^* \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}_2} \text{[Red-Cong-Let]}$$

$$\frac{\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}' \qquad \Gamma \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_1' \qquad \Gamma \vdash \mathsf{e}_2 \rhd^* \mathsf{e}_2'}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rhd^* \mathsf{if}\ \mathsf{e}'\ \mathsf{then}\ \mathsf{e}_1'\ \mathsf{else}\ \mathsf{e}_2'} \text{[Red-Cong-If]}$$

$$\frac{\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'}{\Gamma \vdash \mathsf{succ}\ \mathsf{e} \rhd^* \mathsf{succ}\ \mathsf{e}'} \text{[Red-Cong-Succ]}$$

$$\frac{\Gamma \vdash \mathsf{A} \rhd^* \mathsf{A}' \qquad \Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}' \qquad \Gamma \vdash \mathsf{e}_1 \rhd^* \mathsf{e}_1' \qquad \Gamma \vdash \mathsf{e}_2 \rhd^* \mathsf{e}_2'}{\Gamma \vdash \mathsf{indnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2 \rhd^* \mathsf{indnat}\ \mathsf{A}'\ \mathsf{e}'\ \mathsf{e}_1'\ \mathsf{e}_2'} \text{[Red-Cong-ElimNat]}$$

Fig. A17: ECC congruence conversion rules.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\frac{\Gamma \vdash e_1 \rhd^* e \qquad \Gamma \vdash e_2 \rhd^* e}{\Gamma \vdash e_1 \equiv e_2}\,[\equiv]$$

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

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

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

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

$$\frac{}{\Gamma \vdash \mathsf{Prop} \preceq \mathsf{Type}_0}\,[\preceq\text{-}\textsc{Prop}] \qquad \frac{}{\Gamma \vdash \mathsf{Type}_i \preceq \mathsf{Type}_{i+1}}\,[\preceq\text{-}\textsc{Cum}]$$

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

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

Fig. A18: ECC equivalence and subtyping

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_0}\ [\textsc{Ax-Prop}]$$

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

$$\frac{x \overset{\delta}{=} e : A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ [\textsc{Def-Var}] \qquad \frac{\Gamma \vdash e : A \qquad \Gamma, x \overset{\delta}{=} e : A \vdash e' : B}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : B[x := e]}\ [\textsc{Let}]$$

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

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

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

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

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\ e : B[x := \mathsf{fst}\ e]}\ [\textsc{Snd}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Bool} : \mathsf{Type}_0}\ [\textsc{Bool}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ [\textsc{True}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ [\textsc{False}]$$

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

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash e : B}\ [\textsc{Conv}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}_0}\ [\textsc{Nat}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}\ [\textsc{Zero}] \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ e : \mathsf{Nat}}\ [\textsc{Succ}]$$

$$\frac{\begin{array}{c}\Gamma, x : \mathsf{Nat} \vdash A : U \qquad \Gamma \vdash e : \mathsf{Nat} \\ \Gamma \vdash e_1 : A[x := \mathsf{zero}] \qquad \Gamma \vdash e_2 : \Pi\, n : \mathsf{Nat}.\, \Pi\, r : A[x := n].\, A[x := \mathsf{succ}\ n]\end{array}}{\Gamma \vdash \mathsf{indnat}\ A\ e\ e_1\ e_2 : A[x := e]}\ [\textsc{ElimNat}]$$

Fig. A19: ECC typing

$\boxed{\vdash \Gamma}$

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

Fig. A20: ECC well-formed environments.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Universes	$\mathbf{U}$	$::=$	$\mathbf{Prop} \mid \mathbf{Type}_i$
Values	$\mathbf{V}$	$::=$	$\mathbf{x} \mid \mathbf{U} \mid \boldsymbol{\lambda}\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \boldsymbol{\Pi}\, \mathbf{x} : \mathbf{M}.\, \mathbf{M}$
		$\mid$	$\boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{M}.\, \mathbf{M} \mid \langle \mathbf{V}, \mathbf{V} \rangle \mid \mathbf{Bool}$
		$\mid$	$\mathbf{true} \mid \mathbf{false} \mid \mathbf{Nat} \mid \mathbf{zero}$
		$\mid$	$\mathbf{succ}\ \mathbf{V} \mid \mathbf{refl}\ \mathbf{V} \mid \mathbf{V} \equiv \mathbf{V}$
Computations	$\mathbf{N}$	$::=$	$\mathbf{V} \mid \mathbf{V}\ \mathbf{V} \mid \mathbf{fst}\ \mathbf{V} \mid \mathbf{snd}\ \mathbf{V}$
		$\mid$	$\mathbf{indnat}\ \mathbf{M}\ \mathbf{V}\ \mathbf{V}\ \mathbf{V}$
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	$\boldsymbol{\Gamma}$	$::=$	$\cdot \mid \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{V} \mid \boldsymbol{\Gamma}, \mathbf{x} \overset{\delta}{=} \mathbf{N} : \mathbf{N}$

(a) Run-time Syntax

$\mathbf{e}, \mathbf{A}, \mathbf{B}$	$::=$	$\mathbf{x} \mid \mathbf{U} \mid \boldsymbol{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}$
	$\mid$	$\boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{e} \mid \mathbf{e}\ \mathbf{e}$
	$\mid$	$\boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}$
	$\mid$	$\langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}$
	$\mid$	$\mathbf{fst}\ \mathbf{e} \mid \mathbf{snd}\ \mathbf{e}$
	$\mid$	$\mathbf{let}\ \mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{Bool}$
	$\mid$	$\mathbf{true} \mid \mathbf{false}$
	$\mid$	$\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2$
	$\mid$	$\mathbf{Nat} \mid \mathbf{zero} \mid \mathbf{succ}\ \mathbf{e}$
	$\mid$	$\mathbf{indnat}\ \mathbf{A}\ \mathbf{e}\ \mathbf{e}_1\ \mathbf{e}_2$

(b) Typing Syntax

Fig. A21: CC_e^A syntax.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\frac{\Gamma \vdash e : e_1 \equiv e_2}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Reflect}] \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \text{refl } e \equiv \text{refl } e'}\ [\equiv\text{-Cong-Refl}]$$

$$\frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash A' \equiv B'}{\Gamma \vdash (A \equiv A') \equiv (B \equiv B')}\ [\equiv\text{-Cong-Equiv}] \qquad \frac{\Gamma \vdash e \rhd e'}{\Gamma \vdash e \equiv e'}\ [\equiv\text{-Step}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2}{\Gamma \vdash e[x := e_1] \equiv e[x := e_2]}\ [\equiv\text{-Subst}]$$

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

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

$$\frac{\Gamma \vdash e \equiv \text{true}}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \equiv e_1}\ [\equiv\text{-If-}\beta_1] \qquad \frac{\Gamma \vdash e \equiv \text{false}}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \equiv e_2}\ [\equiv\text{-If-}\beta_2]$$

$$\frac{}{\Gamma \vdash \text{if } e' \text{ then } e \text{ else } e \equiv e}\ [\equiv\text{-If2}] \qquad \frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash e \equiv e'}{\Gamma \vdash \lambda\, x : A.\, e \equiv \lambda\, x : A'.\, e'}\ [\equiv\text{-Cong-Lam}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \qquad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash e_1\ e_2 \equiv e_1'\ e_2'}\ [\equiv\text{-Cong-App}]$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \equiv B'}{\Gamma \vdash \Pi\, x : A.\, B \equiv \Pi\, x : A'.\, B'}\ [\equiv\text{-Cong-Pi}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \qquad \Gamma \vdash e_2 \equiv e_2' \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash \langle e_1, e_2 \rangle \text{ as } A \equiv \langle e_1', e_2' \rangle \text{ as } A'}\ [\equiv\text{-Cong-Pair}]$$

$$\frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \text{fst } e \equiv \text{fst } e'}\ [\equiv\text{-Cong-Fst}] \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \text{snd } e \equiv \text{snd } e'}\ [\equiv\text{-Cong-Snd}]$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \equiv B'}{\Gamma \vdash \Sigma\, x : A.\, B \equiv \Sigma\, x : A'.\, B'}\ [\equiv\text{-Cong-Sig}] \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \text{succ } e \equiv \text{succ } e'}\ [\equiv\text{-Cong-Succ}]$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma \vdash e \equiv e' \qquad \Gamma \vdash e_1 \equiv e_1' \qquad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \text{indnat } A\ e\ e_1\ e_2 \equiv \text{indnat } A'\ e'\ e_1'\ e_2'}\ [\equiv\text{-Cong-ElimNat}]$$

$$\frac{\Gamma \vdash e \equiv e' \qquad \Gamma, x \overset{\delta}{=} e : A \vdash e_1 \equiv e_1'}{\Gamma \vdash \text{let } x = e \text{ in } e_1 \equiv \text{let } x = e' \text{ in } e_1'}\ [\equiv\text{-Cong-Let}]$$

$$\frac{\Gamma \vdash e \equiv e' \qquad \Gamma, V \equiv \text{true} \vdash e_1 \equiv e_1' \qquad \Gamma, V \equiv \text{false} \vdash e_2 \equiv e_2'}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \equiv \text{if } e' \text{ then } e_1' \text{ else } e_2'}\ [\equiv\text{-Cong-If}]$$

$$\frac{}{\Gamma \vdash e \equiv e}\ [\equiv\text{-Refl}] \qquad \frac{\Gamma \vdash e' \equiv e}{\Gamma \vdash e \equiv e'}\ [\equiv\text{-Symm}] \qquad \frac{\Gamma \vdash e_1 \equiv e' \qquad \Gamma \vdash e' \equiv e_2}{\Gamma \vdash e_1 \equiv e_2}\ [\equiv\text{-Trans}]$$

Fig. A22: CC_e^A equivalence

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

$\boxed{\mathbf{M} \mapsto \mathbf{M}'}$

$$\begin{array}{rcl}
\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M})\ \mathbf{V}] & \mapsto_\beta & \mathbf{K}\langle\!\langle \mathbf{M}[\mathbf{x} := \mathbf{V}] \rangle\!\rangle \\
\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] & \mapsto_{\sigma_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{snd}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] & \mapsto_{\sigma_2} & \mathbf{K}[\mathbf{V}_2] \\
\mathbf{K}[\mathbf{indnat}\ \mathbf{M}\ \mathbf{zero}\ \mathbf{V}_1\ \mathbf{V}_2] & \mapsto_{\iota_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{indnat}\ \mathbf{M}\ (\mathbf{succ}\ \mathbf{V})\ \mathbf{V}_1\ \mathbf{V}_2] & \mapsto_{\iota_2} & \mathbf{let}\ \mathbf{x}_1 = (\mathbf{V}_2\ \mathbf{V})\ \mathbf{in}\ \mathbf{let}\ \mathbf{x}_2 = (\mathbf{indnat}\ \mathbf{M}\ \mathbf{V}\ \mathbf{V}_1\ \mathbf{V}_2)\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2] \\
\mathbf{let}\ \mathbf{x} = \mathbf{V}\ \mathbf{in}\ \mathbf{M} & \mapsto_\zeta & \mathbf{M}[\mathbf{x} := \mathbf{V}] \\
\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 & \mapsto_{\mathbb{B}_1} & \mathbf{M}_1 \\
\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 & \mapsto_{\mathbb{B}_2} & \mathbf{M}_2
\end{array}$$

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

$$\frac{}{\mathbf{M} \mapsto^* \mathbf{M}}\ [\textsc{Red-Refl}] \qquad \frac{\mathbf{M} \mapsto \mathbf{M}_1 \qquad \mathbf{M}_1 \mapsto^* \mathbf{M}'}{\mathbf{M} \mapsto^* \mathbf{M}'}\ [\textsc{Red-Trans}]$$

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

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

Fig. A23: CC_e^A evaluation.

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

$$\begin{array}{rcl}
\mathbf{K}\langle\!\langle \mathbf{N} \rangle\!\rangle & \stackrel{\text{def}}{=} & \mathbf{K}[\mathbf{N}] \\
\mathbf{K}\langle\!\langle \mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{M} \rangle\!\rangle & \stackrel{\text{def}}{=} & \mathbf{let}\ \mathbf{x} = \mathbf{N}'\ \mathbf{in}\ \mathbf{K}\langle\!\langle \mathbf{M} \rangle\!\rangle \\
\mathbf{K}\langle\!\langle \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 \rangle\!\rangle & \stackrel{\text{def}}{=} & \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{K}\langle\!\langle \mathbf{M}_1 \rangle\!\rangle\ \mathbf{else}\ \mathbf{K}\langle\!\langle \mathbf{M}_2 \rangle\!\rangle
\end{array}$$

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

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

Fig. A24: Composition of configurations.

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

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

Fig. A25: CC_e^A continuation typing.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

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

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \begin{array}{c}\Gamma, x : \mathsf{Bool} \vdash B : U \\ \Gamma, p : e \equiv \mathsf{true} \vdash e_1 : B[x := \mathsf{true}] \qquad \Gamma, p : e \equiv \mathsf{false} \vdash e_2 : B[x := \mathsf{false}]\end{array}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : B[x := e]}\ [\text{If}]$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{refl}\ e : e \equiv e}\ [\text{Refl}] \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash A' : \mathsf{Type}_i}{\Gamma \vdash A \equiv A' : \mathsf{Type}_i}\ [\text{Equiv}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_0}\ [\text{Ax-Prop}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ [\text{Ax-Type}]$$

$$\frac{x : A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ [\text{Var}] \qquad \frac{x \overset{\delta}{=} e : A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ [\text{Def-Var}]$$

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

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

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

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

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\ e : B[x := \mathsf{fst}\ e]}\ [\text{Snd}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Bool} : \mathsf{Type}_0}\ [\text{Bool}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ [\text{True}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ [\text{False}]$$

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

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash e : B}\ [\text{Conv}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}_0}\ [\text{Nat}]$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}\ [\text{Zero}] \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ e : \mathsf{Nat}}\ [\text{Succ}]$$

$$\frac{\begin{array}{c}\Gamma, x : \mathsf{Nat} \vdash A : U \qquad \Gamma \vdash e : \mathsf{Nat} \\ \Gamma \vdash e_1 : A[x := \mathsf{zero}] \qquad \Gamma \vdash e_2 : \Pi\, n : \mathsf{Nat}.\, \Pi\, r : A[x := n].\, A[x := \mathsf{succ}\ n]\end{array}}{\Gamma \vdash \mathsf{indnat}\ A\ e\ e_1\ e_2 : A[x := e]}\ [\text{ElimNat}]$$

Fig. A26: CC_e^A typing

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press

Lemma 1.2 (Context Replacement (full definition)).

1. *If* $\vdash \Gamma, x : A, \Gamma'$ *and* $\Gamma \vdash B \preceq A$, *then* $\vdash \Gamma, x : B, \Gamma'$.
2. *If* $\Gamma, x : A, \Gamma' \vdash e : C$ *and* $\Gamma \vdash B \preceq A$, *then* $\Gamma, x : B, \Gamma' \vdash e : C$.
3. *If* $\Gamma, x : A, \Gamma' \vdash C \preceq C'$ *and* $\Gamma \vdash B \preceq A$, *then* $\Gamma, x : B, \Gamma' \vdash C \preceq C'$.
4. *If* $\Gamma, x : A, \Gamma' \vdash e \equiv e'$ *and* $\Gamma \vdash B \preceq A$, *then* $\Gamma, x : B, \Gamma' \vdash e \equiv e'$.
5. *If* $\Gamma, x : A, \Gamma' \vdash e \triangleright^* e'$ *and* $\Gamma \vdash B \preceq A$, *then* $\Gamma, x : B, \Gamma' \vdash e \triangleright^* e'$.
6. *If* $\Gamma, x : A, \Gamma' \vdash e \triangleright e'$ *and* $\Gamma \vdash B \preceq A$, *then* $\Gamma, x : B, \Gamma' \vdash e \triangleright e'$.

Lemma 1.3 (Context Definition Replacement (full definition)).

1. *If* $\vdash \Gamma, x \overset{\delta}{=} e : A, \Gamma'$ *and* $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\vdash \Gamma, x \overset{\delta}{=} e' : A, \Gamma'$.
2. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 : C$, $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\Gamma, x \overset{\delta}{=} e' : A, \Gamma' \vdash e_1 : C$.
3. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash C \preceq C'$ *and* $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\Gamma, x \overset{\delta}{=} e' : A, \Gamma' \vdash C \preceq C'$.
4. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 \equiv e_2$ *and* $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\Gamma, x \overset{\delta}{=} e' : A, \Gamma' \vdash e_1 \equiv e_2$.
5. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 \triangleright^* e_2$ *and* $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\Gamma, x \overset{\delta}{=} e' : A, \Gamma' \vdash e_1 \triangleright^* e_2$.
6. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 \triangleright e_2$ *and* $\Gamma \vdash e' : A$ *and* $\Gamma \vdash e \equiv e'$, *then* $\Gamma, x \overset{\delta}{=} e' : A, \Gamma' \vdash e_1 \triangleright e_2$.

Lemma 1.4 (Cut (full definition)).

1. *If* $\vdash \Gamma, x : A, \Gamma'$ *and* $\Gamma \vdash e : A$, *then* $\vdash \Gamma, \Gamma'[x := e]$.
2. *If* $\Gamma, x : A, \Gamma' \vdash e' : C$ *and* $\Gamma \vdash e : A$, *then* $\Gamma, \Gamma'[x := e] \vdash e'[x := e] : C[x := e]$.
3. *If* $\Gamma, x : A, \Gamma' \vdash C \preceq C'$ *and* $\Gamma \vdash e : A$, *then* $\Gamma, \Gamma'[x := e] \vdash C[x := e] \preceq C'[x := e]$.
4. *If* $\Gamma, x : A, \Gamma' \vdash e_1 \equiv e_2$ *and* $\Gamma \vdash e : A$, *then* $\Gamma, \Gamma'[x := e] \vdash e_1[x := e] \equiv e_2[x := e]$.
5. *If* $\Gamma, x : A, \Gamma' \vdash e_1 \triangleright^* e_2$ *and* $\Gamma \vdash e : A$, *then* $\Gamma, \Gamma'[x := e] \vdash e_1[x := e] \triangleright^* e_2[x := e]$.
6. *If* $\Gamma, x : A, \Gamma' \vdash e_1 \triangleright e_2$ *and* $\Gamma \vdash e : A$, *then* $\Gamma, \Gamma'[x := e] \vdash e_1[x := e] \triangleright e_2[x := e]$.

Lemma 1.5 (Cut With Definitions (full definition)).

1. *If* $\vdash \Gamma, x \overset{\delta}{=} e : A, \Gamma'$, *then* $\vdash \Gamma, \Gamma'[x := e]$.
2. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e' : C$, *then* $\Gamma, \Gamma'[x := e] \vdash e'[x := e] : C[x := e]$.
3. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash C \preceq C'$, *then* $\Gamma, \Gamma'[x := e] \vdash C[x := e] \preceq C'[x := e]$.
4. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 \equiv e_2$, *then* $\Gamma, \Gamma'[x := e] \vdash e_1[x := e] \equiv e_2[x := e]$.
5. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 \triangleright^* e_2$, *then* $\Gamma, \Gamma'[x := e] \vdash e_1[x := e] \triangleright^* e_2[x := e]$.
6. *If* $\Gamma, x \overset{\delta}{=} e : A, \Gamma' \vdash e_1 \triangleright e_2$, *then* $\Gamma, \Gamma'[x := e] \vdash e_1[x := e] \triangleright e_2[x := e]$.

https://doi.org/10.1017/S0956796822000090 Published online by Cambridge University Press