ANF Preserves Dependent Types Up to Extensional Equality

by

Paulette Koronkevich

B.Sc., Indiana University, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

ANF Preserves Dependent Types Up to Extensional Equality

submitted by **Paulette Koronkevich** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

William J. Bowman, Assistant Professor, Computer Science, UBC
Supervisor

Ronald Garcia, Associate Professor, Computer Science, UBC
Supervisory Committee Member

Abstract

A growing number of programmers use dependently typed languages such as Coq to machine-verify important properties of 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, 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.

Lay Summary

Modern programmers write programs using high-level programming languages instead of writing sequences of instructions that a computer understands. The instructions that a computer understands are simple, yet require extreme care and thought to do even basic reasoning. *Compilers* translate high-level programs into instructions understood by a computer. Compilers allow programmers to write programs in a high-level language, which is far more human-readable than instructions understood by a computer. Programmers can also reason about the intended behavior of programs *before* a computer executes the instructions. The instructions should have the same behavior as the high-level program, but compilers themselves are programs that can introduce mistakes. This thesis attempts to rectify the disconnect between the specified behavior of the high-level program and the instructions executed by a computer. In particular, this thesis focuses on one transformation a compiler applies to a high-level program to transform it into a sequence of instructions.

Preface

This thesis is based on unpublished work conducted collaboratively in the Software Practices Lab at the University of British Columbia. None of the text of this thesis is taken directly from previously published or collaborative articles; however, a large majority of the text in Chapters 1, 3-8 come from a submission to the ACM SIGPLAN Principles of Programming Languages (POPL) 2022 conference. I wrote Chapters 2 and 9.

This project originally started at Northeastern University during supervisor Professor William J. Bowman's doctoral work. After a technical problem in the main type preservation proof was discovered after a submission to POPL 2019, I was tasked with fixing the proof. Fixing the proof led to the key insights of this work: the target programming language must be designed to encode semantic equivalences relied upon by compilation, and the proof architecture relies on a key lemma that allows compositional reasoning with a continuation. The design of the target programming language changed, in particular, the equivalence relation changed due to a technical problem discovered while translating the formal models to the Coq proof assistant. The formal models and proofs in Coq are written by Ramon Rakow, a former undergraduate at University of British Columbia. Ramon began the translation to Coq as part of an independent study with Professor William J. Bowman.

The source language ECC of the translation originally did not have natural numbers and their recursive eliminator; I added these features and rewrote all the necessary proofs. I also redesigned the target language CC_e^A to include propositional equalities, equality reflection, and a new equivalence relation. With the redesign of the target language, several theorems including the type-preservation theorem had to be reproven, which was all done by me.

Table of Contents

List of Figures

Acknowledgements

Thank you to everyone who has supported me this thesis.

In particular, I am extremely grateful for my supervisor, William J. Bowman. I could not have asked for a better supervisor in graduate school, and surely could not have done this without you and your support. Thank you for always being there, for listening, and for caring deeply, both about me and this work. I am so fortunate to know you and to work with you, and looking forward to many more years of great work (and fun!) with you.

I am also thankful and appreciative of my other committee member, Ronald Garcia. You are always unwaveringly supportive of me and everyone else in the Software Practices Lab (SPL). I owe you many M & M's for everything you have done for me, especially for the wonderful feedback on this thesis.

Thank you Ramon Rakow, for your constant hard work on the mechanization of the proofs in this work, even after graduating. You are always a joy to work with, and extremely kind and supportive. William and I have never laughed quite as hard in meetings together than in the meetings we had with you.

I am so grateful for my family and friends who have helped me along the way as well. Thank you Joseph Wonsil, for your support and everything else to long to list here, but especially the help with all the technical details of submitting a thesis. Thank you to **every** member of the SPL, as I would have surely not have had such a pleasant experience writing this thesis without you. Thank you especially to Joey Eremondi who provided comments on an earlier submission of this work, as well as Ramon's mother Meg O'Donovan.

While not directly involved in this work, I could not be here if it were not for my undergraduate advisor, Sam Tobin-Hochstadt. You helped me fall in love with computer science and programming languages, and got me to where I am today. I am so lucky to have been supported by you all four years at Indiana University (and beyond!).

We acknowledge the support of the NSERC grant RGPIN-2019-04207. Cette recherche a été financée par le CRSNG, numéro de référence RGPIN-2019-04207.

Chapter 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 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.

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 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 dynamically enforce safety at the boundary between the typed IL and untyped components Ahmed [2015]. Applied to Coq, this technique would provide significant confidence that the executable program P is as correct as the verified program S.

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. These semantic equivalences are syntactic changes to the program that preserve the expected runtime behavior. Past work has used strong axioms, including parametricity and impredicativity, to recover these equivalences Bowman et al. [2018]. However, this approach does not support all dependent type system features, including *higher universes.* Higher universes enable programs to abstract over types, and the types of types, which is useful for generic programming and abstract mathematics. Reliance on such strong axioms prevent programs that use generic programming from being compiled. Restricting the user from compiling these programs is undesirable as then the compiler does not scale to a realistic dependently-typed language. 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.

This leads us to my thesis statement:

> *Extensionality* allows us to prove dependent-type preservation by encoding the semantic equivalences relied upon by compilation in the compiler intermediate language.

Extensionality allows the type system to consider two types (or two terms embedded in a type) *equivalent* if the program contains an expression representing a proof that the two types (or terms) are *equal*. That is, the internal equivalence judgement the type system relies on can consider two terms $\mathbf{e}_1$ and $\mathbf{e}_2$ equivalent if one can construct a proof of equality in the language, which is a term of identity type $\mathbf{e}_1 \equiv \mathbf{e}_2$. Using this feature, the translation can insert hints for the type system about which terms the type system 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 can not handle, and does so without relying on parametricity or impredicativity. Relying on extensionality has one key downside: decidable type checking becomes more complex. We discuss how to mitigate this downside in Chapter 8.

To demonstrate how extensionality can be used to prove dependent-type preservation, we present a dependent-type preserving translation to A-normal form (ANF), a compiler intermediate representation that makes control flow explicit and facilitates optimizations Flanagan et al. [1993], Sabry and Felleisen [1992]. The source language 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 and Asai [2018a]. 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 also provides substantial evidence that a feature like extensionality can be used to prove type preservation of a *practical* dependently typed programming language.

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 equivalence reflection to access these equalities.

To prove dependent-type preservation, we not only rely on extensionality in the target language but also develop a new proof architecture for reasoning about the ANF translation. This proof architecture is necessary because 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 be 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.

Although this thesis focuses on showing how extensionality can be used to prove dependent-type preservation of the ANF translation specifically, it also provides insights into dependent-type preservation for other translations. The target IL is designed to express and check semantic equivalences that the translation relies on for correctness. This lesson likely extends to other translations in a type-preserving compiler for a dependently typed programming language.

Chapter 2

Background

In this chapter we introduce the basics of dependent types with simple examples and a summary of how to define a dependently typed programming language. Then we introduce the idea of formally defining compilation and linking, the general statement of type preservation, and the ANF translation. We conclude by discussing the general structure of a proof of type preservation for a dependently typed programming language compiler.

2.1 Dependent Types

For a typical typed programming language, one can think of terms and types of a program as existing in separate syntactic categories. Terms can be thought of as the syntax used for computation, used to *define data* and the operations on that data. Types can be thought of as the syntax used to specify and *describe data* defined by terms in the program. For example, the term for a list of natural numbers (containing the numbers 1 and 2) may be written as (cons 1 (cons 2 empty)). To describe this program term with a type, a type like list nat is useful, as it specifies that the term is a list containing natural numbers. These types list and nat are separate syntactic constructs used to describe these program terms, such as cons and 1. The benefit of having types in a programming language is to aid programmer reasoning about the expected behavior of programs, as well as allowing the compiler to check certain properties *before* the programs execute.

Types in a programming language usually describe simple properties about terms, but a programmer may want their types to describe additional details about program terms. For example, a programmer may want to specify a list of natural numbers *with a particular length*. However, the list type described earlier is a *type constructor* that only requires another type to specify the types of the elements in the list, such as list nat which specifies the elements of the list have type nat. Writing something like list nat 2 to describe a natural number list of length 2 is usually not possible, as the types and terms in a typical typed programming language are separate syntactic constructs.

Dependent types can describe additional details about terms by allowing type constructors to include program terms. Instead of the typical type constructor list, one can use a *dependent list type*, a type constructor that requires a type to describe the elements of the list *and* a program term to describe the length of the list [1]. For example, one can describe a list of natural numbers of length one with the dependent type list nat 1. Notice that the type constructor depends on the program term 1 to describe the length of the list. A list of length two is not described by this type.

Including program terms in a type allows one to write detailed specifications of the desired behavior of programs and statically prevent some errors. As an example, consider a function first that expects to get the first element from a list of natural numbers, of type list nat $\rightarrow$ nat. If the function first is applied to a list with no elements, this results in a run-time error. However, using dependent types, we can statically prevent such an error by specifying that first takes a natural number n and a *dependent* list instead, with length greater than or equal to one:

$$\forall\ \mathsf{n} : \mathsf{nat}\ .\ \mathsf{list\ nat}\ (\mathsf{n} + 1) \rightarrow \mathsf{nat}$$

This type specifies that the list has length greater than or equal to one by using a *dependent function type* ($\forall$) instead of a non-dependent function type ($\rightarrow$). Dependent function types name the argument passed to the function, so that the output type (the type after the dot .) can refer to the specific *term* (of the specified input type) passed to the function. In first, the variable n specifically refers to the first

[1]This is usually called a *dependent vector* in dependently typed programming languages, and usually uses the same nil and cons constructors as the list type.

argument (a term) of any application of first. We then use this variable n in the type of the list, which we specify must have a length of n + 1, depending on this particular n passed to first. This ensures that the list has at least length one, as n is a nat and thus is either zero or greater. In this way, dependent types can statically prevent the erroneous use of first on any list without a first element.

As seen in the first example, one way to include program terms in a type is by binding variables, as is done by the dependent function type (∀). Then, substituting actual program terms for these variables creates a more precise type, in comparison to the type without the variables substituted with terms. For example, in the type for first, a substitution occurs when applying first to its first argument, as the first argument is bound to the variable n. For example, the type of the application (first 0) is: list nat (0 + 1) $\rightarrow$ nat. The type of the dependent list changes from list nat (n + 1) to list nat (0 + 1), as the variable n has been replaced with the concrete term 0. This substitution creates a more precise type by specifying that the term (first 0) requires an input term of type list nat (0 + 1) as opposed to an input term of type list nat (n + 1), where n is arbitrary.

Not only can dependently typed programming languages allow programmers to write detailed specifications of their programs, but they also allow programmers to *prove* that a program meets its specification. Proofs are possible in dependently typed programming languages due to the *Curry-Howard correspondence* Howard [1980], which allows one to view programs as proofs and vice versa. For example, a proof of an implication $A \rightarrow B$ can be viewed as a function $\lambda x : A.\ e$ of type $A \rightarrow B$. The proof of an implication $A \rightarrow B$ typically follows a structure such as "assuming proposition A, prove proposition B." In a similar manner, a function $\lambda x : A.\ e$ can be *checked* to have type $A \rightarrow B$, by assuming the variable x has type A when checking the body e has type B. In other words, "assuming x has type A, check (prove) e has type B." Since building a proof follows the process of building a well-typed term so closely, checking proofs can now be replaced with type checking terms in a dependently typed language. Thus proofs *about* our programs in a dependently typed language can themselves be a program in the same language! In this way, programmers can use a single language to write specifications for programs, write the programs themselves, and prove properties about these programs.

Now that we have covered the basic ideas behind dependent types, where types can rely on terms and vice versa, we briefly describe what is required to formally define a dependently typed programming language. When presenting a dependently typed programming language formally in this thesis, we first describe the syntax of the types and terms, and then present the typing rules. These typing rules show which program terms are described by a dependent type, such as a dependent list, and which terms have a type where a substitution occurs, such as a dependent function.

In most type systems, the typing rules rely on *subtyping* to compare types to determine if some type A is "similar enough" to some other type B. Subtyping is used to determine if terms of type A can be used instead of terms of type B. For example, suppose a function f requires an int type that describes integers, but is applied to a term 0 of type nat. Intuitively, we know that the application (f 0) is well typed, as terms of type nat could be included as terms of type int. The subtyping rules allow this application to be well typed, as nat is a subtype of int, i.e., all natural numbers are integers.

Subtyping for dependent type systems change from most type systems, as now terms can be substituted into types. For example, how can one determine if a type list nat m is a subtype of list nat n for some arbitrary program terms m and n? Since these terms can now exist in types, the subtyping rules must include some notion of determining whether program terms m and n are *equivalent*.

Equivalence of terms is often determined by "executing" them. We define a reduction relation to determine if terms in our type system will reduce ("execute") to the same term. Defining this reduction relation is often done by first defining rules for certain small terms, called the *single-step reduction rules*. For example, one such single-step reduction rule could state that for any term n, the program $n + 0$ steps to n, written as $n + 0 \triangleright n$. Then, the reduction relation is defined as reducing any subterms of a term that match the left-hand side of a single-step reduction rule. The reduction relation indicates that the single-step reduction rules can be applied to any possible matching subterms, for any number of times. We use this reduction relation when determining of a type A is a subtype of B, as these types may contain terms. We determine if these terms within the types A and B are equivalent by using the reduction relation.

In summary, when presenting a dependently typed language (two in total in this thesis), we present:

- The syntax of types and terms
- The typing rules, which are reliant on the
- The subtyping rules, which are reliant on the
- Equivalence relation, which is determined by the
- Reduction relation, which is determined by the
- Single-step reduction rules for terms.

2.2 Compilers

In this section, we introduce compilation and linking, and the formal notation for defining both. We then discuss correctness of a compiler, how correctness can be violated by linking errors, and how linking errors can be prevented by using a type-preserving compiler. Finally, we conclude this section by discussing the ANF translation, the particular compiler pass presented in this thesis.

A compiler can be thought of as a function that takes in a program and transforms it into another program. The input program is referred to as a *source program*, and the resulting program is referred to as a *target program*. When defining a compiler, one should specify the *source programming language* the compiler is defined over, and the *target programming language* the compiler emits. In this work, we introduce the source and target programming languages, and then the compiler (the translation function) from source to target.

We use a non-bold blue sans-serif font to typeset the source language, and a **bold red serif font** for the target language. The fonts are distinguishable in black-and-white, but the thesis is easier to read when viewed in color. We then define the compiler as a pair of brackets (called semantic brackets) $[\![\mathsf{e}]\!] = \mathbf{e}$, where e is a source expression and $\mathbf{e}$ is the resulting compiled target expression.

A target component $\mathbf{e}$, the output of a compiler, might be *linked* with additional components to execute successfully. For example, the component $\mathbf{e}$ might be linked with system libraries specific to the machine. The linking process can be modeled as providing a *substitution* γ that is a map of free variables to their definitions. For example, a compiled component $\mathbf{e}$ may rely on some system function $\mathbf{f}$ to execute, thus $\mathbf{e}$ must be linked with the system library to gain access to the definition of $\mathbf{f}$. The system library function names and definitions would appear in the subtitution γ, including $\mathbf{f}$ and its definition. We then model linking the target program $\mathbf{e}$ with the substitution by using the notation $\gamma(\mathbf{e})$. This indicates that any definitions in the substitution γ are now "combined" with the target program $\mathbf{e}$ to make a final, whole program.

While a compiler is usually defined over and returns syntax, there is an assumption that the compiler preserves the "behavior" of the source program. One could define a compiler $[\![\mathsf{e}]\!] = \mathbf{42}$, but this would not be a very useful compiler. We need a notion of correctness of a compiler. Correctness is usually expressed in terms of evaluation. If a source program e evaluates to some value v, then the result of compilation $\mathbf{e}$ should also evaluate to some (related) value $\mathbf{v}$. This then requires a relation between values of the two programming languages.

Even with a correct compiler, an error not observed in the source program e is still possible in the target program $\mathbf{e}$ due to linking. For example, say we prove that since the target component $\mathbf{e}$ was compiled with a correct compiler, it has the expected behavior as the source component e. However, suppose $\mathbf{e}$ must be linked with a substitution γ containing definitions not compiled by the correct compiler. A definition in the substitution γ may cause an error in the expected behavior of $\mathbf{e}$.

Type-preserving compilation can be used to prevent these kinds of linking errors. Type-preserving compilation ensures that if a source program e has type A, then the compiled program $[\![\mathsf{e}]\!]$ has type $[\![\mathsf{A}]\!]$, the result of compiling the type A. We can then prevent some linking errors by ensuring the substitution γ that the target program $\mathbf{e}$ links with does not violate any of the guarantees provided by the source program's original types A. That is, given the types of the definitions in the substitution γ, we can ensure that these types do not violate the type of the target program $\mathbf{e}$. To prove a compiler type preserving, as we do in this work, we must prove the statement "if e has type A, then $[\![\mathsf{e}]\!]$ has type $[\![\mathsf{A}]\!]$" about our translation function $[\![\mathsf{e}]\!]$.

2.2.1 The ANF Translation

In this thesis, we focus on proving type preservation of the A-normal form (ANF) translation. ANF is a syntactic form that makes control flow explicit in the syntax of a program Flanagan et al. [1993], Sabry and Felleisen [1992]. 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. This translation is a representative pass for evaluating how extensionality can be used to prove dependent-type preservation, as the pass performs many syntactic changes to the source program. We show how extensionality can be used to encode these semantic equivalences to prove that the ANF translation preserves dependent types.

As an example of ANF, we consider reducing the following term:

$$((1 + 2) + (3 - 2))$$

When reducing this term, we first reduce the subterm $(1 + 2)$ to a value, then the subterm $(3 - 2)$ to a value, then we add these two values. ANF makes this control flow explicit in the syntax by decomposing this term into this series of primitive computations that the machine must execute, sequenced by **let**. Thus, the ANF translation of this term will be:

$$\begin{aligned}
&\mathbf{let}\ \mathbf{x} = \mathbf{(1 + 2)}\ \mathbf{in}\\
&\mathbf{let}\ \mathbf{y} = \mathbf{(3 - 2)}\ \mathbf{in}\\
&\mathbf{x + y}
\end{aligned}$$

Once in ANF, it is simple to formalize a machine semantics to implement evaluation. Each **let**-bound computation $\mathbf{x} = \mathbf{N}$ is some primitive *machine step*, performing the computation $\mathbf{N}$ and binding the value to $\mathbf{x}$. 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).

2.3 Type-Preservation Proofs for Dependent Types

Now that we have covered the basics of dependently typed programming languages, compilers, linking, and type preservation, we discuss the general structure of a type preservation proof for a compiler of a dependently typed programming language. As mentioned previously, when presenting a dependently typed programming language, we present the typing rules, the subtyping rules, the equivalence relation, the reduction relation, and the single-step reduction rules for terms. Since all these components rely upon one another, the type preservation proof for a dependently typed programming language must follow a similar structure.

To prove type preservation "if e has type A, then $[\![\mathsf{e}]\!]$ has type $[\![\mathsf{A}]\!]$", we must also show "if A is a subtype of B, then $[\![\mathsf{A}]\!]$ is a subtype of $[\![\mathsf{B}]\!]$," as the typing rules rely on subtyping. Similarly, since subtyping relies on the equivalence relation, then we must also prove "if $\mathsf{e}_1 \equiv \mathsf{e}_2$, then $[\![\mathsf{e}_1]\!] \equiv [\![\mathsf{e}_2]\!]$." If equivalence relies on reduction relation ($\triangleright^*$), then we must also prove "if $\mathsf{e}_1 \triangleright^* \mathsf{e}_2$, then $[\![\mathsf{e}_1]\!] \equiv [\![\mathsf{e}_2]\!]$." Finally, since the reduction relation relies on the the single-step reduction rules ($\triangleright$), we must prove "if $\mathsf{e}_1 \triangleright \mathsf{e}_2$, then $[\![\mathsf{e}_1]\!] \equiv [\![\mathsf{e}_2]\!]$." Our proof of type-preservation for the ANF translation follows this general structure, with a few additional lemmas, described in the next chapter.

Chapter 3

Main Ideas

The problem with dependent-type preservation has little to do with ANF translation itself, and everything to do with dependent types. Transformations which *ought* to be fine aren't because the type theory is so beholden to details of syntax. This is essentially the problem of *commutative cuts* in type theory Boutillier [2012], Herbelin [2009]. Transformations that change the structure (syntax) 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:

$$\begin{tikzcd}[column sep=small] \mathsf{snd}\ \mathsf{e} \arrow[rr, bend left] & : \mathsf{B}[\mathsf{x} := \mathsf{fst} & \mathsf{e}] \end{tikzcd}$$

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).

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

This is not well typed, even with the following standard dependent-let rule.

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

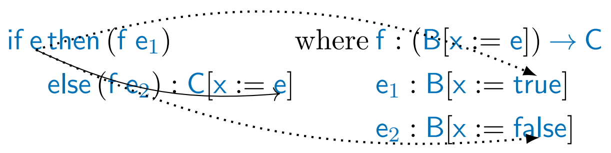
The problem now is the dependent elimination $\mathsf{snd}\,\mathsf{y}$ is let-bound and then used, changing the type to $\mathsf{B}[\mathsf{x} := \mathsf{fst}\,\mathsf{y}]$. This means the equality $\mathsf{y} = \mathsf{e}$ is missing, but is needed to type check this term (indicated by the dotted line 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*, types whose eliminators are regarded as primary, 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*, whose constructors are regarded as primary, 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.

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

Consider an if expression and a function f. The expression f (if e then e_1 else 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), the result is not well typed.



This fails because we need to type check the application f applied to the branches e_1 and e_2. However, now that the application is pushed into the branches, f is expecting an argument of type B[x := e] but is applied to arguments of type B[x := true] and B[x := 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 f 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,Σ, and natural numbers in ECC), and one for positive types (booleans, in ECC).

For negative types, it suffices to use *definitions* Severi and 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:

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

The highlighted part, $x \overset{\delta}{=} e : A$, 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 $A'[x := e]$ 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 and 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*, an equality between two terms, specifically between the term being eliminated and its value. For booleans, we need the following typing rule for **if**. [1]

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

The two highlighted portions of the rule are additions to 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 to definitions, these equalities are accessed through an additional type equivalence rule [≡-Reflect], which states that an equivalence holds between two terms if an equality exists in the environment. This is a simplified and weaker form of the standard propositional equivalence reflection rule, which states that an equivalence holds between two terms if there exists some proof term of their equality.

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

[1]This rule essentially implements the convoy pattern Chlipala [2013], which we discuss further in Chapter 8.

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.

Equivalence reflection is not a perfect solution, as adding this type equivalence rule can make type checking undecidable. We discuss how to recover decidability of CC_e^A in Chapter 8.

Formalizing Type-Preserving ANF Translation Despite these simple features of 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.[2] 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 and 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 $\mathcal{A}\,[\![\mathsf{e}]\!]$ as shorthand for translating e with an empty continuation, $\mathcal{A}\,[\![\mathsf{e}]\!]\,[\cdot]$.

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

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

[2]This is how ANF translation is implemented in Scheme by Flanagan et al. [1993], although their formal model is as a reduction system.

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 $\mathbf{B}$. 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 is used. [3] We discuss this rule further in Chapter 5, and how continuation typing does not extend the target type theory.

The key lemma to prove type preservation is the following.

Lemma 3.1. *If* $\Gamma \vdash e : A$ *and* $\mathcal{A}[\![\Gamma]\!], \Gamma' \vdash \mathbf{K} : (\mathcal{A}[\![e]\!] : \mathcal{A}[\![A]\!]) \Rightarrow \mathbf{B}$*, then* $\mathcal{A}[\![\Gamma]\!], \Gamma' \vdash \mathcal{A}[\![e]\!]\,\mathbf{K} : \mathbf{B}$.

The intuition behind this lemma is 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 Γ' and answer type $\mathbf{B}$. Intuitively, this is because the continuation $\mathbf{K}$, not the expression e, dictates the final answer type $\mathbf{B}$ of the translation, and the environment Γ' provides some additional equalities and definitions under which we type check the transformed e.

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

Chapter 4

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 and Poll [1994]. This language is a subset of CIC and is based on the presentation given in the Coq reference manual[1]; Timany and Sozeau [2017] give a recent account of the metatheory for 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 4.1. ECC extends the Calculus of Constructions (CC) Coquand and Huet [1988] with Σ types (strong dependent pairs) and an infinite predicative hierarchy of universes. There is no explicit phase distinction, *i.e.*, there is no syntactic distinction between *terms*, which represent run-time expressions, and *types*, which classify terms. However, we usually use the meta-variable e to evoke a term, and the meta-variables A and B to evoke a type. The language includes one impredicative universe, Prop, and an infinite hierarchy of predicative universes Type_i. The syntax of expressions e includes names x, universes U, dependent function types $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, functions $\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}$, application $\mathsf{e}_1\ \mathsf{e}_2$, dependent pair types $\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, dependent pairs $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle\ \mathsf{as}\ \Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, first $\mathsf{fst}\ \mathsf{e}$ and second $\mathsf{snd}\ \mathsf{e}$ projections of dependent pairs, dependent let $\mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}'$, the boolean type Bool, the boolean values true and false, dependent if $\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$, the natural number type Nat, the natural

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

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

Figure 4.1: ECC Syntax

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

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

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

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

Figure 4.2: ECC Dynamic Semantics

number values zero and $\mathsf{succ}\ \mathsf{e}$, and the recursive eliminator for natural numbers $\mathsf{elimnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2$. For brevity, we omit the type annotation on dependent pairs, as in $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$. The let-bound definitions do not include type annotations; this is not standard, but type checking is still decidable Severi and Poll [1994], and it simplifies our ANF translation. As is standard, we assume uniqueness of names and consider syntax up to α-equivalence.

In Figure 4.2, we give the reductions $\Gamma \vdash \mathsf{e} \rhd \mathsf{e}'$ for ECC, which are entirely standard, and we elide the environment Γ from the rules as it does not change.

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

$$\frac{}{\Gamma \vdash e \rhd^* e}\ [\textsc{Red-Refl}] \qquad \frac{\Gamma \vdash e \rhd e_1 \qquad \Gamma \vdash e_1 \rhd^* e'}{\Gamma \vdash e \rhd^* e'}\ [\textsc{Red-Step}]$$

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

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

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

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2'}{\Gamma \vdash e_1\ e_2 \rhd^* e_1'\ e_2'}\ [\textsc{Red-Cong-App}]$$

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

Figure 4.3: ECC Congruence Conversion Rules

We extend reduction to conversion by defining $\Gamma \vdash e \rhd^* e'$ to be the reflexive, transitive, compatible closure of reduction $\rhd$. The conversion relation, defined in Figure 4.3 and Figure 4.4, is used to compute equivalence between types, but we can also view it as the operational semantics for the language. We define $\text{eval}(e)$ as the evaluation function for whole-programs using conversion, which we use in our compiler correctness proof.

In Figure 4.5 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

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

$$\frac{\Gamma, x \overset{\delta}{=} N : A \vdash M \rhd^* M'}{\Gamma \vdash \mathsf{let}\, x = N \,\mathsf{in}\, M \rhd^* \mathsf{let}\, x = N \,\mathsf{in}\, M'} \text{ [Red-Cong-Let1]}$$

$$\frac{\Gamma \vdash N \rhd^* N' \qquad \Gamma, x \overset{\delta}{=} N' : A \vdash M \rhd^* M'}{\Gamma \vdash \mathsf{let}\, x = N \,\mathsf{in}\, M \rhd^* \mathsf{let}\, x = N' \,\mathsf{in}\, M'} \text{ [Red-Cong-Let2]}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash A \rhd^* A' \\ \Gamma \vdash e \rhd^* e' \qquad \Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2'\end{array}}{\Gamma \vdash \mathsf{elimnat}\, A\; e\; e_1\; e_2 \rhd^* \mathsf{elimnat}\, A'\; e'\; e_1'\; e_2'} \text{ [Red-Cong-ElimNat]}$$

Figure 4.4: ECC Congruence Conversion Rules (continued)

universes. Subtyping is not completely contravariant for Π types to allow type inclusions to be modeled as set inclusions in a set-theoretic model Luo [1990].

We define the type system for ECC in Figure 4.7 and Figure 4.8, which is mutually defined with well-formedness of environments in Figure 4.6. The typing rules are entirely standard for a dependent type system. 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}. We have impredicative function types in Prop, given by [Prod-Prop]. The rules for application, [App], second projection, [Snd], let, [Let], if, [If], and the eliminator for natural numbers [ElimNat] substitute sub-expressions into the type system. These are the typing rules that introduce difficulty in type-preserving compilation for dependent types.

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

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

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

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

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

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

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

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

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

Figure 4.5: ECC Equivalence and Subtyping

$\boxed{\vdash \Gamma}$

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

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

Figure 4.6: ECC Well-Formed Environments

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

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

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

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

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

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B}\;[\textsc{Lam}] \qquad \frac{\Gamma \vdash e : \Pi\, x : A'.\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\; e' : B[x := e']}\;[\textsc{App}]$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x : A \vdash B : \mathsf{Type}_i}{\Gamma \vdash \Sigma\, x : A.\, B : \mathsf{Type}_i}\;[\textsc{Sig}]$$

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

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

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

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

Figure 4.7: ECC Typing

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}_0}\ [\textsc{Nat}] \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}\ [\textsc{Zero}] \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ e : \mathsf{Nat}}\ [\textsc{Succ}]$$

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

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

Figure 4.8: ECC Typing (continued)

Chapter 5

Target: ECC with ANF Support

Our target language, CC_e^A, is a variant of ECC with a modified typing rule for dependent **if** that introduces propositional equalities between terms and equivalence reflection for accessing assumed equalities. The type system of CC_e^A is not particularly novel as its type theory is adapted from the extensional Calculus of Constructions Oury [2005]. While 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, as type checking relies on the reduction relation for ECC, and thus any reduced terms must be converted to ANF. Instead, we show that our compiler generates only ANF restricted terms in CC_e^A, and define a separate ANF-preserving machine-like semantics for evaluating programs in ANF. We typeset CC_e^A in a **bold, red, serif font**; in later sections, we reserve this font exclusively for the ANF restricted 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 CC_e^A from ECC, or (2) as first embedding ECC in CC_e^A and then rewriting CC_e^A terms into ANF. In Chapter 6 we present the compiler as process (1), a compiler from ECC to ANF 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.

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

(a) Run-time Syntax

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

(b) Typing Syntax

Figure 5.1: CC_e^A Syntax

We give the ANF syntax for CC_e^A in Figure 5.1(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 $\mathbf{V} \equiv \mathbf{V}'$ and $\mathbf{refl}\, \mathbf{V}$ to enable preserving dependent typing for $\mathbf{if}$ expressions and the join-point optimization, further described in Chapter 7. We do not require an elimination form for identity types;

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

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

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

Figure 5.2: CC_e^A Typing (excerpt)

they are instead used via the restricted form of equivalence 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, $(\mathbf{let}\ x = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ x)[N] = (\mathbf{let}\ x = N\ \mathbf{in}\ \mathbf{snd}\ x)$. Since continuations are not first-class objects, we cannot express control effects—continuations are syntactically guaranteed to be used linearly. Despite the *syntactic* distinctions, we still do not enforce a *phase* distinction—configurations (programs) can appear in types. Finally in Figure 5.1(b), we give the full non-ANF syntax, denoted by metavariables 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.

We give the new typing rules in Figure 5.2. The rules for the identity type are standard. The key change in CC_e^A from ECC is in the typing rule for **if**. The typing rule for $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ 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, shown in Figure 5.3 and Figure 5.8. The rule [≡-Reflect] is used for accessing the assumed propositional equalities, such as those introduced in the typing rule for **if**. The rules [≡-Subst] and [≡-Step] state that equivalence is preserved under substitution and the standard reduction defined for ECC. We add congruence rules for all

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

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

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

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

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

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

$$\frac{\Gamma \vdash e \equiv \mathsf{true}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \equiv e_1}\ [\equiv\text{-If-}\eta_1]$$

$$\frac{\Gamma \vdash e \equiv \mathsf{false}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \equiv e_2}\ [\equiv\text{-If-}\eta_2]$$

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

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

Figure 5.3: CC_e^A Equivalence

syntactic forms including the identity type, shown in Figure 5.8. 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 5.4, typically called *renormalization* in the literature Kennedy [2007], Sabry and Wadler [1997]. In ANF, all continuations are left associated, so the standard definition of substitution does not preserve ANF. The

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

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

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

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

Figure 5.4: Composition of Configurations

β-reduction takes an ANF configuration $\mathbf{K}[(\lambda\,\mathbf{x} : \mathbf{A}.\,\mathbf{M})\ \mathbf{V}]$ 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. 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, *i.e.*, substitution. In ANF, as in monadic normal form, substitution can be used to compose any expression with a *value*, since names are values and values can always be replaced by values. But substitution cannot just replace a name, which is a *value*, with a *computation* or *configuration*. That wouldn't be well typed. So how do we compose computations with configurations? We can use $\mathbf{let}$, as in $\mathbf{let}\ \mathbf{y} = \mathbf{N}\ \mathbf{in}\ \mathbf{M}$, which we can imagine

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

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

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

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

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

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

Figure 5.5: CC_e^A Evaluation

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 $\mathbf{M}$ into a continuation $\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathbf{M}'$, since this would result in the non-ANF (but valid monadic) expression $\mathbf{let}\ \mathbf{y} = \mathbf{M}\ \mathbf{in}\ \mathbf{M}'$. Instead, ANF requires a new operation to compose configurations: $\mathbf{K}\langle\langle \mathbf{M} \rangle\rangle$. This operation is more generally known as *hereditary substitution* Pfenning and Davies [2001], Watkins et al. [2003], a form of substitution that maintains canonical forms. So we can think of it as a form of substitution, or, simply, as composition.

In Figure 5.5, we present the call-by-value (CBV) evaluation semantics for ANF CC_e^A terms. It is essentially standard, but recall that β-reduction produces a configuration $\mathbf{M}$ which must be composed with the existing continuation $\mathbf{K}$.

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

$$\frac{}{\Gamma \vdash [\cdot] : (M' : A) \Rightarrow A}\ [\text{K-Empty}]$$

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

Figure 5.6: CC_e^A Continuation Typing

To maintain ANF, the natural number eliminator for the **succ** case must first let-bind the intermediate application and recursive call before plugging the final application into the existing continuation K. This semantics is only for the run-time evaluation of configurations; during type checking, we continue to use the conversion relation defined in Chapter 4.

5.1 Dependent Continuation Typing

The ANF translation manipulates continuations K as independent entities. To reason about them, and thus to reason about the translation, we introduce continuation typing, defined in Figure 5.6. The type $(M' : A) \Rightarrow B$ of a continuation expresses that this continuation expects to be composed with a term equal to the configuration M' of type A and returns a result of type B when completed. Normally, M' is equivalent to some computation N, but it must be generalized to a configuration M' to support typing **if** expressions. This type formally expresses the idea that M' is depended upon in the rest of the computation. For the empty continuation $[\cdot]$, M' is arbitrary since an empty continuation has no "rest of the program" that could depend on anything.

Intuitively, what we want from continuation typing is a compositionality property—that we can reason about the types of configurations $K[N]$ (generally, for configurations $K\langle\langle M\rangle\rangle$) by composing the typing derivations for K and N. To get this property, a continuation type must express not merely the *type* of its hole A, but *which term* N will be bound in the hole. We see this formally from the typing rule [Let] (the same for CC_e^A as for ECC in Chapter 4), since showing that $\mathbf{let}\ \mathbf{y} = \mathbf{N}\ \mathbf{in}\ \mathbf{M}$ is well-typed requires showing that $\mathbf{y} \overset{\delta}{=} \mathbf{N} : \mathbf{A} \vdash \mathbf{M} : \mathbf{B}$, that is, requires knowing the definition $\mathbf{y} \overset{\delta}{=} \mathbf{N} : \mathbf{A}$. 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, Theorem 5.1 (Cut).

The result of a continuation type cannot depend on the term that will be plugged in for the hole, *i.e.*, 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—*i.e.*, is admissible—we prove Theorem 5.1 and Theorem 5.3, 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 Theorem 5.1 (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 5.1 (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 $\mathbf{\Gamma}, \mathbf{y} \overset{\delta}{=} \mathbf{N} : \mathbf{A} \vdash \mathbf{M} : \mathbf{B}$ 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's 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.

Lemma 5.2 (Cut Modulo Equivalence)**.** *If* $\mathbf{\Gamma} \vdash \mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$, $\mathbf{\Gamma} \vdash \mathbf{N} : \mathbf{A}$, $\mathbf{\Gamma} \vdash \mathbf{N}' : \mathbf{A}$, *and* $\mathbf{\Gamma} \vdash \mathbf{N} \equiv \mathbf{N}'$, *then* $\mathbf{\Gamma} \vdash \mathbf{K}[\mathbf{N}'] : \mathbf{B}$.

Proof. By cases on the structure of $\mathbf{K}$. □

Proof. By cases on the structure of $\mathbf{K}$.

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

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

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

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

□

The final lemma about continuation typing is also the key to why ANF is type preserving for $\mathbf{if}$. The heterogeneous composition operations perform the ANF translation on $\mathbf{if}$ expressions by composing the branches with the current continuation, reassociating all intermediate computations. The following lemma states that if a configuration $\mathbf{M}$ is well typed and equivalent to another configuration $\mathbf{M}'$ under an extended environment $\mathbf{\Gamma}'$, and the continuation $\mathbf{K}$ is well-typed with respect to $\mathbf{M}'$ then the composition $\mathbf{K}\langle\langle\mathbf{M}\rangle\rangle$ is well typed. The proof is simple, except for the $\mathbf{if}$ case, which essentially must prove that ANF is type preserving for $\mathbf{if}$.

Lemma 5.3 (Hetereogeneous Cut). *If* $\Gamma \vdash \mathbf{M} : \mathbf{A}$, $\Gamma \vdash \mathbf{M'} : \mathbf{A'}$, *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 Theorem 5.2.

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 $\mathbf{x} \overset{\delta}{=} \mathbf{N} : \mathbf{A}$.

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

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

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

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

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) $\mathbf{B'}[\mathbf{x} := \mathbf{true}]\mathbf{A'}$ under a context $\mathbf{p} : \mathbf{V} \equiv \mathbf{true}$ (analogously for $\mathbf{M}_2$ and $\mathbf{false}$).

For (1), note that $\mathbf{p} : \mathbf{V} \equiv \mathbf{true} \vdash \mathbf{V} \equiv \mathbf{true}$ by [$\equiv$-Reflect].

$$\begin{aligned} \mathbf{M}_1 &\equiv \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 && \text{by } [\equiv\text{-If-}\eta_1] \\ &\equiv \mathbf{M'} && \text{by premise} \end{aligned}$$

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 [≡-Reflect]. □

5.2 Consistency

To demonstrate that the new typing rule for **if** is consistent, we develop a syntactic model of CC_e^A in extensional CIC (eCIC). The model essentially implements the new **if** rule using the *convoy pattern* Chlipala [2013], but leaves the rest of the propositional equalities and equivalence reflection essentially unchanged. Each **if** expression $\mathbf{if}\,\mathrm{e}\,\mathbf{then}\,\mathrm{e}_1\,\mathbf{else}\,\mathrm{e}_2$ 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 equivalence reflection using the translation of Oury [2005], but this is not necessary for consistency.

The essence of the model is given in Figure 5.7. 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 5.7. 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 $\perp$ (the empty type). This means that if CC_e^A were inconsistent, then we could translate the proof of $\perp$ into a proof of $\perp$ in eCIC, but no such proof exists in eCIC, so CC_e^A is consistent.

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

Lemma 5.4 (Model Preserves Empty Type)**.** $[\![\perp]\!]_M \equiv \perp$

The essence of showing both that equivalence is preserved and that typing is preserved is in showing that the auxiliary definitions in Figure 5.7 exist and are well typed.

Lemma 5.5 (Model Preserves Equivalence)**.** *If* $\mathrm{e}_1 \equiv \mathrm{e}_2$ *then* $[\![\mathrm{e}_1]\!]_M \equiv [\![\mathrm{e}_2]\!]_M$.

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

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

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

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

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

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

Figure 5.7: CC_e^A Model in eCIC (excerpts)

Lemma 5.6 (Model Preserves Typing). *If* $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ *then* $[\![\mathbf{\Gamma}]\!]_M \vdash [\![\mathbf{e}]\!]_M : [\![\mathbf{A}]\!]_M$.

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 5.7 (Consistency). *There is no* $\mathbf{e}$ *such that* $\cdot \vdash \mathbf{e} : \bot$

5.3 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, *i.e.*, 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). Theorem 5.8 is the formal statement of naturality in ANF: composing a term $\mathbf{M}$ with its continuation $\mathbf{K}$ in ANF is equivalent to running $\mathbf{M}$ to a value and substituting the result into the continuation $\mathbf{K}$. Formally, this states that composing continuations in ANF is sound with respect to standard substitution. The proof follows by straightforward equational reasoning.

Lemma 5.8 (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: $\mathbf{M} = \mathbf{let\,x = N\,in\,M'}$

Must show that

$\mathbf{let\,x = N'\,in\,K}\langle\langle\mathbf{M'}\rangle\rangle \equiv \mathcal{NN}(\mathbf{K}[\mathbf{let\,x = N'\,in\,M'}])$.

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

Case: $\mathbf{M} = \mathbf{if\,V\,then\,M_1\,else\,M_2}$

Must show that

$\mathbf{if\,V\,then\,K}\langle\langle\mathbf{M_1}\rangle\rangle\,\mathbf{else\,K}\langle\langle\mathbf{M_2}\rangle\rangle \equiv \mathcal{NN}(\mathbf{K}[\mathbf{if\,V\,then\,M_1\,else\,M_2}])$.

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

$\mathbf{let}\ \mathbf{x} = \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2\ \mathbf{in}\ \mathbf{M} \equiv$
$\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M})\ \mathbf{else}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M})$.

$$\begin{aligned}
& \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M})\ \mathbf{else}\ (\mathbf{let}\ \mathbf{x} = \mathbf{M}_1\ \mathbf{in}\ \mathbf{M}) \\
\equiv\ & \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}[\mathbf{x} := \mathbf{M}_1]\ \mathbf{else}\ \mathbf{M}[\mathbf{x} := \mathbf{M}_2] \\
& \text{by } \rhd_\zeta \text{ and } [\equiv\text{-}\textsc{Step}] \\
\equiv\ & \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}[\mathbf{x} := \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2] \\
& \qquad \mathbf{else}\ \mathbf{M}[\mathbf{x} := \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2] \\
& \text{by } [\equiv\text{-}\textsc{If}\text{-}\eta] \text{ and } [\equiv\text{-}\textsc{Subst}] \\
\equiv\ & \mathbf{let}\ \mathbf{x} = (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2)\ \mathbf{in}\ (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}\ \mathbf{else}\ \mathbf{M}) \\
& \text{by } \rhd_\zeta \text{ and } [\equiv\text{-}\textsc{Step}] \\
\equiv\ & \mathbf{let}\ \mathbf{x} = (\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2)\ \mathbf{in}\ \mathbf{M} \\
& \text{by } [\equiv\text{-}\textsc{If2}]
\end{aligned}$$

□

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 5.9 (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. 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 $\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M}_1)\ \mathbf{V}] \equiv \mathbf{K}\langle\langle \mathbf{M}_1[\mathbf{x} := \mathbf{V}]\rangle\rangle$

$$\begin{aligned} & \mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M}_1)\ \mathbf{V}] & \\ \rhd^* \ & \mathcal{N}\mathcal{N}(\mathbf{K}[\mathbf{M}_1[\mathbf{x} := \mathbf{V}]]) & \text{by } \beta \text{ and congruence} \\ \equiv \ & \mathbf{K}\langle\langle \mathbf{M}_1[\mathbf{x} := \mathbf{V}]\rangle\rangle & \text{by Theorem 5.8} \end{aligned}$$

Case: $\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2\rangle] \mapsto_{\pi_1} \mathbf{K}[\mathbf{V}_1]$

Must show that $\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2\rangle] \equiv \mathbf{K}[\mathbf{V}_1]$, which follows by $\rhd_{\pi_1}$ and congruence. ☐

Theorem 5.10 (Evaluation soundness). $\vdash \mathbf{eval}(\mathbf{M}) \equiv \mathbf{M}$

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \text{succ}\, e \equiv \text{succ}\, e'}\ [\equiv\text{-Cong-Succ}]$$

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

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

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

Figure 5.8: CC_e^A Equivalence (continued)

Chapter 6

ANF Translation

The ANF translation is presented in Figure 6.1. 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 Chapter 5. When translating a value such as x, $\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}$, or $\mathsf{Type}\,_i$, 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 6.1). The proof is straightforward, and thus elided.

Theorem 6.1 (ANF)**.** *For all* e *and* $\mathbf{K}$, $\mathcal{A}\,[\![\mathsf{e}]\!]\;\mathbf{K} = \mathbf{M}$ *for some* $\mathbf{M}$.

Our goal is to prove type preservation: if e is well-typed in the source, then $\mathcal{A}\,[\![\mathsf{e}]\!]$ 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. We proceed top-down, starting from our main theorem, in order to motivate where each lemma comes into play.

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 $\mathbf{K}$.

$\boxed{\mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{K} = \mathbf{M}}$

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

Figure 6.1: Naïve ANF Translation

Theorem 6.2 (Type Preservation). *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *then* $\mathcal{A}\,[\![\Gamma]\!] \vdash \mathcal{A}\,[\![\mathsf{e}]\!] : \mathcal{A}\,[\![\mathsf{A}]\!]$.

Proof. By Theorem 6.3, it suffices to show that $\mathcal{A}[\![\Gamma]\!] \vdash [\cdot] : (\mathcal{A}[\![e]\!] : \mathcal{A}[\![A]\!]) \Rightarrow \mathcal{A}[\![A]\!]$, which follows by [K-Empty]. $\square$

To see why we need a stronger induction hypothesis, consider the $\mathsf{snd}\ e$ case of ANF translation: $\mathcal{A}[\![\mathsf{snd}\ e]\!]\ \mathbf{K} \overset{\text{def}}{=} \mathcal{A}[\![e]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x}]$. The induction hypothesis for Theorem 6.2 proves that $\mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A.\, B]\!]$, 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 $\mathcal{A}[\![e]\!]$. The new continuation also composes the original continuation $\mathbf{K}$ with 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 $\mathcal{A}[\![\mathsf{snd}\ e]\!]$.

We abstract this pattern into Theorem 6.3, 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 $\mathbf{\Gamma'}$. Intuitively, this extended environment $\mathbf{\Gamma'}$ 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 6.3.

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

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

For all sub-expressions e' of type A', $\mathcal{A}[\![e']\!] : \mathcal{A}[\![A']\!]$ holds by instantiating the induction hypothesis with the empty continuation $[\cdot] : (\mathcal{A}[\![e']\!] : \mathcal{A}[\![A']\!]) \Rightarrow \mathcal{A}[\![A']\!]$. 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 Theorem 5.3. This requires proving that $\mathbf{K}'$ is composed with a configuration $\mathbf{M}$ equivalent to $\mathcal{A}[\![e]\!]$, and showing their types are equivalent. To show $\mathbf{M}$ is equivalent to $\mathcal{A}[\![e]\!]$, we use the lemmas Theorem 6.4 and Theorem 5.8 to allow for composing configuations and continuations. Additionally, since several typing rules subsitute sub-expressions into the type system, we use Theorem 6.5 in these cases to show the types of $\mathbf{M}$ and $\mathcal{A}[\![e]\!]$ are equivalent. Each non-value proof case focuses on showing these two equivalences. We show the full proof details in Appendix A.

Case: [Ax-Prop]

We must show that $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}' \vdash \mathbf{K}[\mathbf{Prop}] : \mathbf{B}$. This follows from Theorem 5.2.

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

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

$$\begin{aligned}
\mathcal{A}[\![\mathsf{snd}\ e]\!] &\overset{\text{def}}{=} \mathcal{A}[\![e]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_1) \\
&= \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_1 \langle\!\langle \mathcal{A}[\![e]\!] \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let}\ x_1 = \mathcal{A}[\![e]\!]\ \mathsf{in}\ \mathbf{snd}\ x_1 && \text{by Theorem 5.8} \\
&\equiv \mathbf{snd}\ \mathcal{A}[\![e]\!] \equiv \mathbf{snd}\ \mathbf{x}_1 && \text{by } \triangleright_\zeta \text{ and } \triangleright_\delta
\end{aligned}$$

For goal (2), since $\mathcal{A}[\![B'[x := \mathsf{fst}\ e]]\!] \equiv \mathcal{A}[\![B']\!][\mathbf{x} := \mathcal{A}[\![\mathsf{fst}\ e]\!]]$ by Theorem 6.5, we focus on showing $\mathbf{fst}\ \mathcal{A}[\![e]\!] \equiv \mathcal{A}[\![\mathsf{fst}\ e]\!]$. Focusing on $\mathcal{A}[\![\mathsf{fst}\ e]\!]$, we have:

$$\mathcal{A}[\![\mathsf{fst}\ e]\!] \overset{\text{def}}{=} \mathcal{A}[\![e]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x}_1)$$

$= \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x_1} \langle\langle \mathcal{A}\,[\![\mathsf{e}]\!] \rangle\rangle$ by Theorem 6.4

$\equiv \mathsf{let}\ \mathsf{x_1} = \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{fst}\ \mathsf{x_1}$ by Theorem 5.8

$\triangleright_\zeta\ \mathbf{fst}\ \mathcal{A}\,[\![\mathsf{e}]\!]$

Case: [ElimNat] Following our general structure, we must show

(1) $\mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2} \equiv \mathcal{A}\,[\![\mathsf{elimnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e_1}\ \mathsf{e_2}]\!]$ in an environment where

$\mathbf{y} \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e}]\!] : \mathcal{A}\,[\![\mathsf{Nat}]\!]\,, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e_1}]\!] : \mathcal{A}\,[\![\mathsf{A}[\mathsf{zero} := \mathsf{x}]]\!]\,,$
$\mathbf{x_2} \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e_2}]\!] : \mathcal{A}\,[\![\Pi\,\mathsf{n} : \mathsf{Nat}.\,\Pi\,\mathsf{x}' : \mathsf{A}[\mathsf{x} := \mathsf{n}].\,\mathsf{A}[\mathsf{x} := \mathsf{succ}\ \mathsf{n}]]\!]$

and (2) $\mathcal{A}\,[\![\mathsf{A}]\!][\mathbf{x} := \mathbf{y}] \equiv \mathcal{A}\,[\![\mathsf{A}[\mathsf{x} := \mathsf{e}]]\!]$.

For goal (1) we focus on $\mathcal{A}\,[\![\mathsf{elimnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e_1}\ \mathsf{e_2}]\!]$:
$\mathcal{A}\,[\![\mathsf{elimnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e_1}\ \mathsf{e_2}]\!]$

$$\overset{\text{def}}{=} \mathcal{A}\,[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2})))$$

$$= \mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2}))\langle\langle \mathcal{A}\,[\![\mathsf{e}]\!] \rangle\rangle$$

by Theorem 6.4

$$\equiv \mathsf{let}\ \mathsf{y} = \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathsf{in}\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x_1}\ \mathbf{x_2}))$$

by Theorem 5.8

$$\triangleright_\zeta\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{x_1}\ \mathbf{x_2}))$$

$$= \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{x_1}\ \mathbf{x_2})\langle\langle \mathcal{A}\,[\![\mathsf{e_1}]\!] \rangle\rangle$$

by Theorem 6.4

$$\equiv \mathsf{let}\ \mathsf{x_1} = \mathcal{A}\,[\![\mathsf{e_1}]\!]\ \mathsf{in}\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{x_1}\ \mathbf{x_2})$$

by Theorem 5.8

$$\triangleright_\zeta\ \mathcal{A}\,[\![\mathsf{e_2}]\!]\ (\mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ \mathbf{x_2})$$

$$= \mathbf{let}\ \mathbf{x_2} = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ \mathbf{x_2}\langle\langle \mathcal{A}\,[\![\mathsf{e_2}]\!] \rangle\rangle$$

by Theorem 6.4

$$\equiv \mathsf{let}\ \mathsf{x_2} = \mathcal{A}\,[\![\mathsf{e_2}]\!]\ \mathsf{in}\ \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathcal{A}\,[\![\mathsf{e_1}]\!]\ \mathbf{x_2}$$

by Theorem 5.8

$\triangleright_\zeta$ $\mathbf{elimnat}\ \mathcal{A}[\![\mathsf{A}]\!]\ \mathcal{A}[\![\mathsf{e}]\!]\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathcal{A}[\![\mathsf{e}_2]\!]$

For goal (2), since $\mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{e}]]\!] \equiv \mathcal{A}[\![\mathsf{A}]\!][\mathbf{x} := \mathcal{A}[\![\mathsf{e}]\!]]$ by Theorem 6.5, and $\mathbf{y} \equiv \mathcal{A}[\![\mathsf{e}]\!]$ by the definition $\mathbf{y} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e}]\!] : \mathcal{A}[\![\mathsf{Nat}]\!]$, we conclude using [$\equiv$-Subst].

Case: [If] Following our general structure, we must show

(1) $\mathcal{A}[\![\mathsf{e}_1]\!] \equiv \mathcal{A}[\![\mathsf{if\ e\ then\ e}_1\ \mathsf{else\ e}_2]\!]$ in an environment where $\mathbf{x} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e}]\!] : \mathbf{Bool}, \mathbf{p} : \mathbf{x} \equiv \mathbf{true}$ and

(2) $\mathcal{A}[\![\mathsf{B}'[\mathsf{x}' := \mathsf{e}]]\!] \equiv \mathcal{A}[\![\mathsf{B}']\!][\mathbf{x}' := \mathbf{true}]$ (and analogously for e_2 where $\mathbf{x} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e}]\!] : \mathbf{Bool}, \mathbf{p} : \mathbf{x} \equiv \mathbf{false}$).

Focusing on $\mathcal{A}[\![\mathsf{if\ e\ then\ e}_1\ \mathsf{else\ e}_2]\!]$ in goal (1), we have:

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{if\ e\ then\ e}_1\ \mathsf{else\ e}_2]\!] \\
&\overset{\text{def}}{=} \mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let\ x} = [\cdot]\ \mathbf{in\ if\ x\ then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]) \\
&= \mathbf{let\ x} = [\cdot]\ \mathbf{in\ if\ x\ then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \langle\!\langle \mathcal{A}[\![\mathsf{e}]\!] \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let\ x} = \mathcal{A}[\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{if}\ \mathsf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!] && \text{by Theorem 5.8} \\
&\triangleright_\zeta\ \mathbf{if}\ \mathcal{A}[\![\mathsf{e}]\!]\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!] \\
&\equiv \mathcal{A}[\![\mathsf{e}_1]\!] && \text{by } [\equiv\text{-If-}\eta_1] \text{ since } \mathcal{A}[\![\mathsf{e}]\!] \equiv \mathbf{true} \text{ by } [\equiv\text{-Reflect}]
\end{aligned}$$

For goal (2), $\mathcal{A}[\![\mathsf{B}']\!][\mathbf{x}' := \mathcal{A}[\![\mathsf{e}]\!]] \equiv \mathcal{A}[\![\mathsf{B}']\!][\mathbf{x}' := \mathbf{true}]$ by Theorem 6.5. Since $\mathcal{A}[\![\mathsf{e}]\!] \equiv \mathbf{true}$ we conclude with [$\equiv$-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 $\mathcal{A}[\![\mathsf{e}]\!]\ \mathbf{K}$. To reason about this term, we need to show an equivalence between the compiler and ANF-composition, so we can reason instead about $\mathbf{K}\langle\!\langle \mathcal{A}[\![\mathsf{e}]\!] \rangle\!\rangle$, a definition we can unroll and we know is correct with respect to evaluation via Theorem 5.8 (Naturality). We prove this equivalence, Theorem 6.4 next. This essentially tells us that our compiler is compositional, i.e., 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 6.4 (Compositionality). $\mathbf{K}'\langle\langle \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{K}\rangle\rangle = \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle$

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

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

Case: $\mathsf{e} = \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$

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

Case: $\mathsf{e} = \mathsf{e}_1\ \mathsf{e}_2$

Must show that

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

The proof follows essentially from the definition of continuation composition.

$$\begin{aligned}&&&\mathbf{K}'\langle\langle(\mathcal{A}\,[\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ (\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]))))\rangle\rangle\\ &=&&(\mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{K}'\langle\langle(\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ (\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])))\rangle\rangle)\\ &&&\hfill\text{by the induction hypothesis applied to } \mathsf{e}_1\\ &=&&(\mathcal{A}\,[\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{K}'\langle\langle(\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])\rangle\rangle))\\ &&&\hfill\text{by definition of continuation composition}\\ &=&&(\mathcal{A}\,[\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ (\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{K}'\langle\langle\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]\rangle\rangle)))\\ &&&\hfill\text{by the induction hypothesis applied to } \mathsf{e}_2\\ &=&&(\mathcal{A}\,[\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ (\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle[\mathbf{x}_1\ \mathbf{x}_2])))\\ &&&\hfill\text{by definition of continuation composition}\end{aligned}$$

Case: $\mathsf{e} = \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$ Must show that

$$\begin{aligned}&\mathbf{K}'\langle\langle(\mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K}))\rangle\rangle\\ &= (\mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle))\end{aligned}$$

The proof follows essentially from the definition of continuation composition.

$$\begin{aligned}
& \mathbf{K}'\langle\langle(\mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K}))\rangle\rangle\\
= \quad & (\mathcal{A}[\![\mathsf{e}]\!]\ \mathbf{K}'\langle\langle(\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K})\rangle\rangle)\\
& \text{by the induction hypothesis applied to } \mathsf{e}\\
= \quad & (\mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}'\langle\langle\mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K}\rangle\rangle))\\
& \text{by definition of continuation composition}\\
= \quad & (\mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathbf{K}'\langle\langle\mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}\rangle\rangle\ \mathbf{else}\ \mathbf{K}'\langle\langle\mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K}\rangle\rangle))\\
& \text{by definition of continuation composition}\\
= \quad & (\mathcal{A}[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![\mathsf{e}_1]\!]\ \mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle\ \mathbf{else}\ \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle))\\
& \text{by induction on } \mathsf{e}_1 \text{ and } \mathsf{e}_2
\end{aligned}$$

□

Corollary 6.4.1. $\mathbf{K}\langle\langle\mathcal{A}[\![\mathsf{e}]\!]\rangle\rangle = \mathcal{A}[\![\mathsf{e}]\!]\ \mathbf{K}$

Similarly, at times we need to reason about the translation of terms or types that have a variable substituted, e.g. $\mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{e}']]\!]$. We often use induction on a judgement $\Gamma \vdash \mathsf{e} : \mathsf{A}$, 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 $\mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{e}']]\!]$, 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, Theorem 6.5, 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 shift from the **target language font** to the source language font whenever we shift out of ANF, such as when we perform standard substitution or conversion. When the shift in font is not apparent, we use the language boundary term $\mathcal{NN}()$. This lemma relies on uniqueness of names.

Lemma 6.5 (Substitution). $\mathcal{A}[\![e[x := e']]\!]\ \mathbf{K} \equiv \mathcal{NN}((\mathcal{A}[\![e]\!]\ \mathbf{K})[\mathbf{x} := \mathcal{A}[\![e']\!]])$

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

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

$$\begin{aligned}
&\mathcal{NN}(\mathcal{A}[\![x]\!]\ \mathbf{K}[\mathbf{x} := \mathcal{A}[\![e']\!]]) \\
&= \mathcal{NN}(\mathbf{K}[\mathbf{x}][\mathbf{x} := \mathcal{A}[\![e']\!]]) \\
&= \mathcal{NN}(\mathbf{K}[\mathcal{A}[\![e']\!]]) \\
&\equiv \mathbf{K}\langle\!\langle \mathcal{A}[\![e']\!] \rangle\!\rangle && \text{by Theorem 5.8} \\
&\equiv \mathcal{A}[\![e']\!]\ \mathbf{K} && \text{by Theorem 6.4}
\end{aligned}$$

Case: $e = \mathsf{Prop}$ Trivial.

Case: $e = \Pi\, x' : A.\, B$

Must show that $\mathcal{A}[\![\Pi\, x' : A.\, B[x := e']]\!]\ \mathbf{K} \equiv \mathcal{NN}((\mathcal{A}[\![\Pi\, x' : A.\, B]\!]\ \mathbf{K})[x := \mathcal{A}[\![e']\!]])$

$$\mathcal{A}[\![\Pi\, x' : A.\, B[x := e']]\!]\ \mathbf{K}$$

$$= \mathcal{A}[\![\Pi\, x' : A[x := e'].\, B[x := e']]\!]\ \mathbf{K}$$

by substitution

$$= \mathbf{K}[\mathbf{\Pi\, x'} : \mathcal{A}[\![A[x := e']]\!].\, \mathcal{A}[\![B[x := e']]\!]]$$

by translation

$$\equiv \mathbf{K}[\mathbf{\Pi\, x'} : \mathcal{NN}(\mathcal{A}[\![A]\!][\mathbf{x} := \mathcal{A}[\![e']\!]]).\, \mathcal{NN}(\mathcal{A}[\![B]\!][\mathbf{x} := \mathcal{A}[\![e']\!]])]$$

by the induction hypothesis

$$= \mathcal{NN}(\mathbf{K}[\mathbf{\Pi\, x'} : \mathcal{A}[\![A]\!].\, \mathcal{A}[\![B]\!]][\mathbf{x} := \mathcal{A}[\![e']\!]])$$

by definition of substitution

$$= \mathcal{NN}((\mathcal{A}[\![\Pi\, x' : A.\, B]\!]\ \mathbf{K})[\mathbf{x} := \mathcal{A}[\![e']\!]])$$

by definition

Case: $\mathsf{e} = \mathsf{e}_1\ \mathsf{e}_2$

Must show that $\mathcal{A}\,[\![(\mathsf{e}_1\ \mathsf{e}_2)[\mathbf{x} := \mathsf{e}']]\!]\ \mathbf{K} \equiv \mathcal{N}\mathcal{N}((\mathcal{A}\,[\![\mathsf{e}_1\ \mathsf{e}_2]\!]\ \mathbf{K})[\mathbf{x} := \mathcal{A}\,[\![\mathsf{e}']\!]])$

$\mathcal{A}\,[\![(\mathsf{e}_1\ \mathsf{e}_2)[\mathsf{x} := \mathsf{e}']]\!]\ \mathbf{K}$

$= \mathcal{A}\,[\![\mathsf{e}_1[\mathsf{x} := \mathsf{e}']\ \mathsf{e}_2[\mathsf{x} := \mathsf{e}']]\!]\ \mathbf{K}$

by substitution

$= \mathcal{A}\,[\![\mathsf{e}_1[\mathsf{x} := \mathsf{e}']]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e}_2[\mathsf{x} := \mathsf{e}']]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]$

by translation

$\equiv \mathcal{A}\,[\![\mathsf{e}_1[\mathsf{x} := \mathsf{e}']]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}$
$\quad \mathcal{N}\mathcal{N}((\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])[\mathbf{x} := \mathcal{A}\,[\![\mathsf{e}']\!]])$

by IH applied to e_1

$\equiv \mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2][\mathsf{x} := \mathcal{A}\,[\![\mathsf{e}']\!]][\mathbf{x} := \mathcal{A}\,[\![\mathsf{e}']\!]]$

by IH applied to e_2

$= \mathcal{N}\mathcal{N}((\mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])[\mathbf{x} := \mathcal{A}\,[\![\mathsf{e}']\!]])$

by substitution

$= \mathcal{N}\mathcal{N}((\mathcal{A}\,[\![\mathsf{e}_1\ \mathsf{e}_2]\!]\ \mathbf{K})[\mathbf{x} := \mathcal{A}\,[\![\mathsf{e}']\!]])$

by substitution

□

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 6.6. *If* $\Gamma \vdash \mathsf{e} \preceq \mathsf{e}'$ *then* $\mathcal{A}\,[\![\Gamma]\!] \vdash \mathcal{A}\,[\![\mathsf{e}]\!] \preceq \mathcal{A}\,[\![\mathsf{e}']\!]$

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

Case: [$\preceq$-$\equiv$]. Follows by Theorem 6.7.

Case: [$\preceq$-Trans]. Follows the induction hypothesis.

Case: [$\preceq$-Prop]. Trivial, since $\mathcal{A}\,[\![\mathsf{Prop}]\!] = \mathbf{Prop}$ and $\mathcal{A}\,[\![\mathsf{Type}_0]\!] = \mathbf{Type}_0$.

Case: [$\preceq$-Cum]. Trivial, since $\mathcal{A}[\![\mathsf{Type}_i]\!] = \mathbf{Type}_i$ and $\mathcal{A}[\![\mathsf{Type}_{i+1}]\!] = \mathbf{Type}_{i+1}$.

Case: [$\preceq$-Pi].

We must show that $\mathcal{A}[\![\Gamma]\!] \vdash \mathcal{A}[\![\Pi\, x_1 : A_1.\, B_1]\!] \preceq \mathcal{A}[\![\Pi\, x_2 : A_2.\, B_2]\!]$

By definition of the translation, we must show $\mathcal{A}[\![\Gamma]\!] \vdash \mathbf{\Pi}\, \mathbf{x}_1 : \mathcal{A}[\![A_1]\!].\, \mathcal{A}[\![B_1]\!] \preceq \mathbf{\Pi}\, \mathbf{x}_2 : \mathcal{A}[\![A_2]\!].\, \mathcal{A}[\![B_2]\!]$.

If we lift the continuations in type annotations A_1 and A_2 outside the Π, as CBPV suggests we should, we would need a new subtyping rule that allows subtyping **let** expressions. As it is, we proceed by [$\preceq$-Pi].

It suffices to show that

(a) $\mathcal{A}[\![\Gamma]\!] \vdash \mathcal{A}[\![A_1]\!] \equiv \mathcal{A}[\![A_2]\!]$, which follows by Theorem 6.7.

(b) $\mathcal{A}[\![\Gamma]\!], \mathbf{x}_1 : \mathcal{A}[\![A_2]\!] \vdash \mathcal{A}[\![B_1]\!] \preceq \mathcal{A}[\![B_2]\!][\mathbf{x}_2 := \mathbf{x}_1]$, which follows by the induction hypothesis.

Case: [$\preceq$-Sig]. Similar to previous case.

□

Subtyping relies on type equivalence, so we 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 6.7. *If* $\Gamma \vdash e \equiv e'$ *then* $\mathcal{A}[\![\Gamma]\!] \vdash \mathcal{A}[\![e]\!] \equiv \mathcal{A}[\![e']\!]$

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

Case: [$\equiv$] Follows by Theorem 6.9.

Case: [$\equiv$-η_1]

By Theorem 6.9, we know $\mathcal{A}[\![e]\!] \equiv \mathcal{A}[\![\lambda\, x : A.\, e_1]\!]$. By transitivity, it suffices to show $\mathcal{A}[\![\lambda\, x : A.\, e_1]\!] \equiv \mathcal{A}[\![e']\!]$.

By [$\equiv$-η_1], since $\mathcal{A}[\![\lambda\, x : A.\, e_1]\!] = \boldsymbol{\lambda}\, \mathbf{x} : \mathcal{A}[\![A]\!].\, \mathcal{A}[\![e_1]\!]$, it suffices to show that $\mathcal{A}[\![e_1]\!] \equiv \mathcal{A}[\![e']\!]\ x_2$

$$\begin{aligned}
&\mathcal{A}\,[\![\mathsf{e}_1]\!] \\
&\equiv \mathcal{A}\,[\![\mathsf{e}'\ \mathsf{x}_2]\!] && \text{by the induction hypothesis} \\
&= \mathcal{A}\,[\![\mathsf{e}']\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{x}_1\ \mathbf{x}_2 \\
&= (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{x}_1\ \mathbf{x}_2)\langle\!\langle \mathcal{A}\,[\![\mathsf{e}']\!]\ \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let}\ \mathsf{x}_1 = \mathcal{A}\,[\![\mathsf{e}']\!]\ \ \mathsf{in}\ \mathsf{x}_1\ \mathsf{x}_2 && \text{by Theorem 5.8} \\
&\equiv \mathcal{A}\,[\![\mathsf{e}']\!]\ \mathsf{x}_2 && \text{by } \triangleright_\zeta \text{ and } [\equiv\text{-Step}]
\end{aligned}$$

Case: $[\equiv\text{-}\eta_2]$ Essentially similar to the previous case. ☐

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.

Lemma 6.8. *If* $\Gamma \vdash \mathsf{e} \triangleright \mathsf{e}'$ *then* $\mathcal{A}\,[\![\Gamma]\!] \vdash \mathcal{A}\,[\![\mathsf{e}]\!] \equiv \mathcal{A}\,[\![\mathsf{e}']\!]$.

Proof. By cases on $\Gamma \vdash \mathsf{e} \triangleright \mathsf{e}'$. We give the key cases.

Case: $\Gamma \vdash \mathsf{x} \triangleright_\delta \mathsf{e}'$

We must show that $\mathcal{A}\,[\![\Gamma]\!] \vdash \mathcal{A}\,[\![\mathsf{x}]\!] \equiv \mathcal{A}\,[\![\mathsf{e}']\!]$

We know that $\mathsf{x} \overset{\delta}{=} \mathsf{e}' : \mathsf{A} \in \Gamma$, and by definition $\mathbf{x} \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e}']\!] : \mathcal{A}\,[\![\mathsf{A}]\!] \in \mathcal{A}\,[\![\Gamma]\!]$, so the goal follows by $[\equiv\text{-}\delta]$.

Case: $\Gamma \vdash \lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1\ \mathsf{e}_2 \triangleright_\beta \mathsf{e}_1[\mathsf{x} := \mathsf{e}_2]$

We must show $\mathcal{A}\,[\![\Gamma]\!] \vdash \mathcal{A}\,[\![(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1)\ \mathsf{e}_2]\!] \equiv \mathcal{A}\,[\![\mathsf{e}_1[\mathsf{x} := \mathsf{e}_2]]\!]$

$$\begin{aligned}
&\mathcal{A}\,[\![\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1\ \mathsf{e}_2]\!] \\
&= \mathcal{A}\,[\![\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{x}_1\ \mathbf{x}_2 \\
&= \mathbf{let}\ \mathbf{x}_1 = (\boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}\,[\![\mathsf{A}]\!].\,\mathcal{A}\,[\![\mathsf{e}_1]\!])\ \mathbf{in}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{x}_1\ \mathbf{x}_2 \\
&\equiv \mathcal{A}\,[\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}\,[\![\mathsf{A}]\!].\,\mathcal{A}\,[\![\mathsf{e}_1]\!]\ \ \mathbf{x}_2
\end{aligned}$$

by $\triangleright_\zeta$ and [$\equiv$-Step]

$= \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}[\![\mathsf{A}]\!].\, \mathcal{A}[\![\mathsf{e}_1]\!])\ \mathbf{x}_2 \langle\!\langle \mathcal{A}[\![\mathsf{e}_2]\!] \rangle\!\rangle$

by Theorem 6.4

$\equiv \mathsf{let}\ \mathsf{x}_2 = \mathcal{A}[\![\mathsf{e}_2]\!]\ \mathsf{in}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}[\![\mathsf{A}]\!].\, \mathcal{A}[\![\mathsf{e}_1]\!])\ \mathsf{x}_2$

by Theorem 5.8

$\equiv (\boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}[\![\mathsf{A}]\!].\, \mathcal{A}[\![\mathsf{e}_1]\!])\ \mathcal{A}[\![\mathsf{e}_2]\!]$

by $\triangleright_\zeta$ and [$\equiv$-Step]

$\equiv \mathcal{NN}(\mathcal{A}[\![\mathsf{e}_1]\!][\mathbf{x} := \mathcal{A}[\![\mathsf{e}_2]\!]])$

by $\triangleright_\beta$ and [$\equiv$-Step]

$\equiv \mathcal{A}[\![\mathsf{e}_1[\mathsf{x} := \mathsf{e}_2]]\!]$

by Theorem 6.5

□

Lemma 6.9. *If* $\Gamma \vdash \mathsf{e} \triangleright^* \mathsf{e}'$ *then* $\mathcal{A}[\![\Gamma]\!] \vdash \mathcal{A}[\![\mathsf{e}]\!] \equiv \mathcal{A}[\![\mathsf{e}']\!]$

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

Case: [Red-Refl], trivial.

Case: [Red-Trans], by Theorem 6.8, the induction hypothesis, and [$\equiv$-Trans].

Case: [Red-Cong-Let]

We have $\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}_1 \triangleright^* \mathsf{let}\ \mathsf{x} = \mathsf{e}'\ \mathsf{in}\ \mathsf{e}_2$, $\Gamma \vdash \mathsf{e} \triangleright^* \mathsf{e}'$, and $\Gamma, \mathsf{x} \overset{\delta}{=} \mathsf{e} : \mathsf{A} \vdash \mathsf{e}_1 \triangleright^* \mathsf{e}_2$.

We must show that $\mathcal{A}[\![\Gamma]\!] \vdash \mathcal{A}[\![\mathsf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}]\!] \equiv \mathcal{A}[\![\mathsf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}']\!]$.

This follows by induction and [$\equiv$-Cong-Let]. □

6.1 Separate Compilation

We also 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.

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

$$\mathsf{true} \approx \mathbf{true} \qquad \mathsf{false} \approx \mathbf{false} \qquad \mathsf{zero} \approx \mathbf{zero} \qquad \frac{\mathsf{v} \approx \mathbf{V}}{\mathsf{succ}\,\mathsf{v} \approx \mathbf{succ}\,\mathbf{V}}$$

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

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

Figure 6.2: Separate Compilation Definitions

We define an independent specification relating observation across languages, which allows us to understand the correctness theorem without reading the compiler. We define the relation $\mathsf{v} \approx \mathbf{V}$ to compare ground values in Figure 6.2.

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

Correctness of separate compilation says that we can either link then run a program in the source language semantics, *i.e.*, using the conversion semantics, or separately compile the term and its closing substitution then run in the ANF evaluation semantics. Either way, we get equivalent terms. The proof is straightforward from the compositionality properties and forward simulations we proved for type preservation.

Theorem 6.10 (Correctness of Separate Compilation)**.** *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$, *(and* A *ground) and* $\Gamma \vdash \gamma$ *then* $\mathbf{eval}(\mathcal{A}[\![\gamma]\!]\,(\mathcal{A}[\![\mathsf{e}]\!])) \approx \mathsf{eval}(\gamma(\mathsf{e}))$.

Proof. The following diagram commutes, because $\equiv$ corresponds to $\approx$ on ground types, the translation commutes with substitution, and preserves equivalence.

$$\begin{tikzcd}
\mathsf{eval}(\gamma(\mathsf{e})) \arrow[r, "\equiv"] \arrow[d, "\equiv"] & \mathcal{A}\llbracket\gamma(\mathsf{e})\rrbracket \arrow[d, "\equiv"] \\
\mathbf{eval}(\mathcal{A}\llbracket\gamma(\mathcal{A}\llbracket\mathsf{e}\rrbracket)\rrbracket) \arrow[r, "\equiv"] & \mathcal{A}\llbracket\gamma(\mathcal{A}\llbracket\mathsf{e}\rrbracket)\rrbracket
\end{tikzcd}$$ $\square$

Chapter 7

Join-Point Optimization

Recall from Figure 5.4 that the composition of a continuation $\mathbf{K}$ with an **if** configuration, $\mathbf{K}\langle\langle \mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{M}_1\ \mathbf{else}\ \mathbf{M}_2 \rangle\rangle$, duplicates $\mathbf{K}$ in the branches:

$$\mathbf{if}\ \mathbf{V}\ \mathbf{then}\ \mathbf{K}\langle\langle \mathbf{M}_1 \rangle\rangle\ \mathbf{else}\ \mathbf{K}\langle\langle \mathbf{M}_2 \rangle\rangle$$

Similarly, the ANF translation in Figure 6.1 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 Chapter 6 to use the definition in Figure 7.1, and prove it is still correct and type preserving. Additionally, 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.

Lemma 7.1.

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

Proof. By mutual induction on $\vdash \Gamma$ and $\Gamma \vdash \mathsf{e} : \mathsf{A}$.

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

$$\vdots$$

$$\begin{aligned}
&\mathcal{A}\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rrbracket\, \mathbf{K} \overset{\text{def}}{=} \\
&\quad \mathbf{let}\ \mathbf{f} = \boldsymbol{\lambda}\, \mathbf{y} : \mathcal{A}\llbracket \mathsf{B}[\mathsf{x} := \mathsf{e}] \rrbracket .\, \boldsymbol{\lambda}\, \mathbf{p} : \mathbf{y} \equiv \mathcal{A}\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rrbracket .\, \mathbf{K}[\mathbf{y}] \\
&\qquad \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e} \rrbracket\, \mathbf{let}\ \mathbf{x}' = [\cdot] \\
&\qquad\qquad \mathbf{in}\ \mathbf{if}\ \mathbf{x}'\ \mathbf{then}\ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\, (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ (\mathbf{f}\ \mathbf{x}_1\ (\mathbf{refl}\ \mathbf{x}_1))) \\
&\qquad\qquad\qquad \mathbf{else}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket\, (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ (\mathbf{f}\ \mathbf{x}_2\ (\mathbf{refl}\ \mathbf{x}_2))) \\
&\qquad \text{where } \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}[\mathsf{x} := \mathsf{e}]
\end{aligned}$$

Figure 7.1: Join-Point Optimized ANF Translation

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) $\mathbf{x}_1 : \mathcal{A}\llbracket \mathsf{A}[\mathsf{x} := \mathsf{e}] \rrbracket$ and (2) $\mathbf{refl}\ \mathbf{x}_1 : \mathbf{x}_1 \equiv \mathcal{A}\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rrbracket$ (analogously for $\mathbf{x}_2$ in the $\mathbf{false}$ branch). (1) follows from the equivalence $\mathcal{A}\llbracket \mathsf{A}[\mathsf{x} := \mathsf{e}] \rrbracket \equiv \mathcal{A}\llbracket \mathsf{A}[\mathsf{x} := \mathsf{true}] \rrbracket$, which has been shown before in Theorem 6.3. (2) follows if we show $(\mathbf{x}_1 \equiv \mathbf{x}_1) \equiv (\mathbf{x}_1 \equiv \mathcal{A}\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rrbracket)$ in an environment where $\mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket : \mathcal{A}\llbracket \mathsf{B}'[\mathsf{x} := \mathsf{true}] \rrbracket$.

By [≡-Cong-Equiv], we must show $\mathbf{x}_1 \equiv \mathcal{A}\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rrbracket$, which by definition is $\mathcal{A}\llbracket \mathsf{e}_1 \rrbracket \equiv \mathcal{A}\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \rrbracket$, previously shown in Theorem 6.3.

□

Chapter 8

Future and Related Work

We first discuss adding recursion and inductive datatypes to our source language ECC, and conjecture that our proof technique can scale to these features. 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.

8.1 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 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 and Ahmed [2018b] 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 size types, but adding size types to Coq is still a work in progress Chan and 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, 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 $\mathcal{A}\,[\![\mathsf{B}[\mathsf{z} := \mathsf{e}'][\mathsf{x} := \mathsf{e}]]\!]$ but then is pushed into the branches and translated with something of type $\mathcal{A}\,[\![\mathsf{B}[\mathsf{z} := \mathsf{e}'_i][\mathsf{x} := \mathsf{C}_i\ \mathsf{y}_i]]\!]$. 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:

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

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

We could show that the ANF translation is type preserving by Theorem 5.2, and by showing that the target fix expression $\mathbf{fix}\, \mathbf{f}(\mathbf{x} : \mathcal{A}\,[\![A]\!]).\, \mathcal{A}\,[\![e]\!]$ 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.

8.2 Recovering Decidability

Our target language CC_e^A is extensional type theory, which is well known to have undecidable type checking. However, intuitively the target terms constructed by the compiler should be decidable. This is because the terms are constructed from terms in a source language with decidable type checking. 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 and Rahul [2001], the inference rules are represented as a higher-order logic program, and the proof checker is a non-deterministic logic program 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 by 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.

8.3 CPS Translation

ANF is favored as a compiler intermediate representation, although not universally. Maurer et al. [2017] argue for ANF over alternatives such as CPS, because ANF makes control flow explicit but keeps evaluation order implicit, automatically avoids administrative redexes, simplifies many optimizations, and keeps code in direct style. Kennedy [2007] argues the opposite—that CPS is preferred to ANF—primarily due to the complexity of the composition operations and join points, and summarizes the arguments for and against.

Most recent work on CPS translation of dependently typed languages focuses on expressing control effects Cong and Asai [2018a,b], Miquey [2017], Pédrot [2017]. When expressing control effects with dependent types, it is *necessary* for consistency to prevent certain dependencies from being expressed Barthe and Uustalu [2002], Herbelin [2005], so these translations do not try to recover dependencies in the way we discuss in Chapter 3.

Two CPS translations exist that do try to recover dependencies in the absence of effects, but fail to scale to higher universes. 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 Theorem 5.8 (Naturality). Unfortunately, this rule relies on interpreting all functions as parametric, and their type translation does not scale to higher universes. Formally, preservation of subtyping, Theorem 6.6 does not hold when extending their CPS translation to a language with higher universes. By contrast, our ANF translation works with higher universes and CC_e^A is orthogonal to parametricity. Cong and Asai [2018a] 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. Their translation is also unable to scale to higher universes.

8.4 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 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], but critically, avoids the central difficulties introduced in Chapter 3. The reason is essentially that monadic form is more compositional than ANF, so dependency is not disrupted in the same way.

Recall from Chapter 5 that our definition of composition was entirely motivated by the need to compose configurations and continuations. In CBPV, and 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.

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

Since `let` can bind the "configuration" $\mathcal{A}[\![\mathsf{e}]\!]$, the typing rule [Let] and the compositionality lemma suffice to show type preservation, without any reasoning about definitions. In fact, we don't 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 and Uustalu [2002].

Recently, Pédrot and 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 Theorem 5.8, but should hold even when the language is extended with effects. The authors note that the thunkability extensions seems to require that the target of the thunkable translation be an extensional type theory.

Monadic normal form has been studied for compilation Benton et al. [1998], and our observations about work on CBPV suggest that monadic normal form may be *even more well-behaved* than ANF for dependent type theory. However, monadic form is in a sense less low-level than ANF. Monadic form still relies on algebraic nesting of expressions (syntactically), or on a stack (in its reduction semantics), where ANF does not except to support returning from a procedure. So while type preservation to monadic form may be simpler, type preservation *from* monadic form to a lower level language may involve some of the same challenges we solve in this work.

Chapter 9

Conclusion

In this thesis, we showed how extensionality allows us to prove dependent-type preservation by encoding the semantic equivalences relied upon by compilation in the compiler intermediate language. We demonstrated this by developing 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. The target language for the translation, CC_e^A, included propositional equalities and the [≡-Reflect] rule to access these equalities. These propositional equalities encode semantic equivalences and allow these equivalences to be threaded through typing derivations. With these equivalences available, we can prove dependent-type preservation by using a proof technique that allows compositional reasoning with the type of the source expression and the type of the target context.

This thesis demonstrates that future dependent-type preserving translations require a way to encode these semantic equivalences in the target IL. This requires identifying *which* semantic equivalences a compiler pass relies on, and how best to encode them. In this work, the trickiest equivalence to encode was between programs with if expressions where the surrounding context is pushed into the branches. In related work such as the closure conversion pass Bowman and Ahmed [2018a], the target IL is also designed to include these semantic equivalences by adding equivalences between closures.

This translation is one compiler pass closer to showing that type-preserving compilation can support all of dependent type theory. Following the work of System F to TAL Morrisett et al. [1999], a complete model of a type-preserving compiler has four additional passes after ANF: closure conversion, hoisting, allocation, and finally code generation to get to an assembly language. The closure conversion pass has already been proven dependent-type preserving Bowman and Ahmed [2018a], and we conjecture that this pass can easily be combined with the ANF translation developed in this thesis. The hoisting pass is simple, as all functions are closed and should be able to be hoisted to the top without difficulty. This leaves the allocation and code generation passes to prove type preserving.

The allocation pass seems the trickiest in the context of dependent types. The main problems we foresee are taming mutable references and allowing necessary cycles in the heap without creating an unsound logic. Combining mutable state or cycles, separately, with dependent types is a difficult problem, and requires a solution before a completing the design of a type-preserving compiler to assembly. Despite the potential difficulties, I look forward to addressing these problems to complete a type-preserving compiler to assembly, perhaps in a future thesis.

Bibliography

D. Ahman. *Fibred Computational Effects*. PhD thesis, University of Edinburgh, Oct. 2017. URL http://arxiv.org/abs/1710.02594.

A. Ahmed. Verified compilers for a multi-language world. In *Summit oN Advances in Programming Languages (SNAPL)*, volume 32, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.15. URL http://drops.dagstuhl.de/opus/volltexte/2015/5013.

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

A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2), Apr. 2015. doi: 10.1145/2701415.

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

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

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

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

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

W. J. Bowman and A. Ahmed. Typed closure conversion for the calculus of constructions. In *International Conference on Programming Language Design and Implementation (PLDI)*, June 2018a. doi: 10.1145/3192366.3192372. URL https://www.williamjbowman.com/papers#cccc.

W. J. Bowman and A. Ahmed. Parametric closure conversion for cic. 2018b. URL https://williamjbowman.com/resources/wjb2018-techreport-parametric-cc-cic.pdf.

W. J. Bowman, Y. Cong, N. Rioux, and A. Ahmed. Type-preserving CPS translation of Σ and Π types is not not possible. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL), Jan. 2018. doi: 10.1145/3158110. URL https://www.williamjbowman.com/papers#cps-sigma.

J. Chan and W. J. Bowman. Practical sized typing for coq, 2020. URL https://arxiv.org/abs/1912.05601.

A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013. ISBN 978-0-262-02665-9. URL http://adam.chlipala.net/cpdt/.

Y. Cong and K. Asai. Handling delimited continuations with dependent types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(ICFP), Sept. 2018a. doi: 10.1145/3236764. URL https://sites.google.com/site/youyoucong212/icfp2018.

Y. Cong and K. Asai. Shifting and resetting in the calculus of constructions. In *International Symposium on Trends in Functional Programming (TFP)*, Apr. 2018b. URL https://sites.google.com/site/youyoucong212/tfp2018.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3), Feb. 1988. doi: 10.1016/0890-5401(88)90005-3.

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

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

R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

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

H. Herbelin. On a few open problems of the calculus of inductive constructions and on their practical consequences, Sept. 2009. URL pauillac.inria.fr/~herbelin/talks/cic.pdf. Updated 2010.

W. A. Howard. The formulae-as-types notion of construction. In H. Curry, H. B., S. J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

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

X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4), Nov. 2009. doi: 10.1007/s10817-009-9155-4.

P. B. Levy. *Call-By-Push-Value*. 2012. ISBN 978-94-007-0954-6. URL https://www.worldcat.org/oclc/7330961929.

Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, July 1990. URL http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-118/.

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

É. Miquey. A classical sequent calculus with dependent types. In *European Symposium on Programming (ESOP)*, Apr. 2017. doi: 10.1007/978-3-662-54434-1_29. URL https://hal.inria.fr/hal-01519929v2.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3), May 1999. doi: 10.1145/319301.319345.

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

G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In C. Hankin and D. Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 142–154. ACM, 2001. doi: 10.1145/360204.360216.

N. Oury. Extensionality in the calculus of constructions. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2005. doi: 10.1007/11541868_18. URL https://doi.org/10.1007/11541868_18.

P. Pédrot. A parametric CPS to sprinkle CIC with classical reasoning. In *Workshop on Syntax and Semantics of Low-Level Languages*, June 2017. URL http://www.cs.bham.ac.uk/~zeilbern/lola2017/abstracts/LOLA_2017_paper_5.pdf.

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

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, Aug. 2001. ISSN 0960-1295. doi: 10.1017/S0960129501003322. URL https://doi.org/10.1017/S0960129501003322.

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

A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6), Nov. 1997. doi: 10.1145/267959.269968.

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

H. Thielecke. From control effects to typed continuation passing. In *Symposium on Principles of Programming Languages (POPL)*, 2003. doi: 10.1145/640128.604144.

A. Timany and M. Sozeau. Consistency of the predicative calculus of cumulative inductive constructions (pcuic). *arXiv preprint arXiv:1710.03912*, 2017. URL https://arxiv.org/abs/1710.03912.

M. Vákár. *In Search of Effectful Dependent Types*. PhD thesis, Oxford University, 2017. URL http://arxiv.org/abs/1706.07997.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In *International Workshop on Types for Proofs and Programs (TYPES)*, 2003. doi: 10.1007/978-3-540-24849-1_23.

Appendix A

Detailed Proofs

In this appendix we include the full proof details of the type preservation proof. We also include the full details of the [If] case of the type preservation proof for the join point optimization.

Lemma A.1.

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

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.

For all sub-expressions e' of type A', $\mathcal{A}[\![e']\!] : \mathcal{A}[\![A']\!]$ holds by instantiating the induction hypothesis with the empty continuation $[\cdot] : (\mathcal{A}[\![e']\!] : \mathcal{A}[\![A']\!]) \Rightarrow \mathcal{A}[\![A']\!]$. 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 Theorem 5.3. This requires proving that $\mathbf{K'}$ is composed with a configuration $\mathbf{M}$ equivalent to $\mathcal{A}[\![e]\!]$, and showing their types are equivalent. To show $\mathbf{M}$ is equivalent to $\mathcal{A}[\![e]\!]$, we use the lemmas Theorem 6.4 and Theorem 5.8 to allow for composing configuations and continuations. Additionally, since several typing rules subsitute sub-expressions into the type system, we use Theorem 6.5 in these cases to show the types of $\mathbf{M}$ and $\mathcal{A}\llbracket e \rrbracket$ are equivalent. Each non-value proof case focuses on showing these two equivalences.

Case: [Ax-Prop]

We must show that $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \mathbf{K}[\mathbf{Prop}] : \mathbf{B}$. This follows from Theorem 5.2.

Case: [Lam]

Must show $\mathcal{A}\llbracket \Gamma \rrbracket \vdash \mathbf{K}[\boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}\llbracket A' \rrbracket .\, \mathcal{A}\llbracket e' \rrbracket] : \mathbf{B}$. This follows from Theorem 5.2 if we can show $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \boldsymbol{\lambda}\,\mathbf{x} : \mathcal{A}\llbracket A' \rrbracket .\, \mathcal{A}\llbracket e' \rrbracket : \mathbf{\Pi}\,\mathbf{x} : \mathcal{A}\llbracket A' \rrbracket .\, \mathcal{A}\llbracket B' \rrbracket$. This follows from [Lam] if we can show $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x} : \mathcal{A}\llbracket A' \rrbracket \vdash \mathcal{A}\llbracket e' \rrbracket : \mathcal{A}\llbracket B' \rrbracket$. This follows by induction on the judgment $\Gamma, x : A' \vdash e' : B'$ with the empty continuation $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x} : \mathcal{A}\llbracket A' \rrbracket \vdash [\cdot] : (\mathcal{A}\llbracket e' \rrbracket : \mathcal{A}\llbracket B' \rrbracket) \Rightarrow \mathcal{A}\llbracket B' \rrbracket$.

Case: [App]

Must show that
$\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \mathcal{A}\llbracket e_1 \rrbracket\, (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket e_2 \rrbracket\, (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])) : \mathbf{B}$.

Let $\mathbf{K}_1 = (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket e_2 \rrbracket\, (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]))$. Our conclusion follows by induction on $\Gamma \vdash e_1 : \Pi\, x : A' .\, B'$ if we show $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \mathbf{K}_1 : (\mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket) \Rightarrow \mathbf{B}$. By [K-Bind], we must show $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket$ and $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket \vdash \mathcal{A}\llbracket e_2 \rrbracket\, (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]) : \mathbf{B}$.

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

The second goal also follows by induction on $\Gamma \vdash e_2 : A'$, but we must show $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket \vdash \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2] : (\mathcal{A}\llbracket e_2 \rrbracket : \mathcal{A}\llbracket A' \rrbracket) \Rightarrow \mathbf{B}$. By [K-Bind] again, we must show $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket \vdash \mathcal{A}\llbracket e_2 \rrbracket : \mathcal{A}\llbracket A' \rrbracket$ (which again follows by induction with the empty continuation) and $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket, \mathbf{x}_2 \overset{\delta}{=} \mathcal{A}\llbracket e_2 \rrbracket : \mathcal{A}\llbracket A' \rrbracket \vdash \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2] : \mathbf{B}$.

This follows from Theorem 5.3 if we can show (1) $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}', \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\llbracket e_1 \rrbracket : \mathcal{A}\llbracket \Pi\, x : A' .\, B' \rrbracket, \mathbf{x}_2 \overset{\delta}{=} \mathcal{A}\llbracket e_2 \rrbracket : \mathcal{A}\llbracket A' \rrbracket \vdash \mathbf{x}_1\ \mathbf{x}_2 : \mathcal{A}\llbracket B'[x := e_2] \rrbracket$,

(2) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_1}]\!] : \mathcal{A}[\![\Pi\, \mathsf{x} : \mathsf{A'}.\, \mathsf{B'}]\!], \mathbf{x_2} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{A'}]\!] \vdash \mathcal{A}[\![\mathsf{e_1}\ \mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{B'}[\mathsf{x} := \mathsf{e_2}]]\!]$, and
(3) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_1}]\!] : \mathcal{A}[\![\Pi\, \mathsf{x} : \mathsf{A'}.\, \mathsf{B'}]\!], \mathbf{x_2} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{A'}]\!] \vdash \mathcal{A}[\![\mathsf{e_1}\ \mathsf{e_2}]\!] \equiv \mathbf{x_1}\ \mathbf{x_2}$.

By $\rhd_\delta$ and [≡-Step], goal (1) changes to $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_1}]\!] : \mathcal{A}[\![\Pi\, \mathsf{x} : \mathsf{A'}.\, \mathsf{B'}]\!], \mathbf{x_2} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{A'}]\!] \vdash \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{B'}[\mathsf{x} := \mathsf{e_2}]]\!]$. We have previously shown that $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathcal{A}[\![\mathsf{e_1}]\!] : \mathbf{\Pi\, x} : \mathcal{A}[\![\mathsf{A'}]\!].\, \mathcal{A}[\![\mathsf{B'}]\!]$ and $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_1}]\!] : \mathcal{A}[\![\Pi\, \mathsf{x} : \mathsf{A'}.\, \mathsf{B'}]\!] \vdash \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{A'}]\!]$. Using these facts with [App], we derive $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_1}]\!] : \mathcal{A}[\![\Pi\, \mathsf{x} : \mathsf{A'}.\, \mathsf{B'}]\!], \mathbf{x_2} \overset{\delta}{=} \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{A'}]\!] \vdash \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathcal{A}[\![\mathsf{e_2}]\!] : \mathcal{A}[\![\mathsf{B'}]\!][\mathbf{x} := \mathcal{A}[\![\mathsf{e_2}]\!]]$. We have that $\mathcal{A}[\![\mathsf{B'}]\!][\mathbf{x} := \mathcal{A}[\![\mathsf{e_2}]\!]] \equiv \mathcal{A}[\![\mathsf{B'}[\mathsf{x} := \mathsf{e_2}]]\!]$ by Theorem 6.5, and derive our conclusion by [Conv].

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

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{e_1}\ \mathsf{e_2}]\!] \\
&\overset{\text{def}}{=} \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathbf{let\ x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}[\![\mathsf{e_2}]\!]\ \mathbf{let\ x_2} = [\cdot]\ \mathbf{in\ x_1\ x_2} \\
&= \mathbf{let\ x_1} = [\cdot]\ \mathbf{in}\ \mathcal{A}[\![\mathsf{e_2}]\!]\ \mathbf{let\ x_2} = [\cdot]\ \mathbf{in\ x_1\ x_2} \langle\!\langle \mathcal{A}[\![\mathsf{e_1}]\!] \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let\ x_1} = \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathsf{in}\ \mathcal{A}[\![\mathsf{e_2}]\!]\ \mathbf{let\ x_2} = [\cdot]\ \mathbf{in}\ \mathsf{x_1}\ \mathbf{x_2} && \text{by Theorem 5.8} \\
&\equiv_\zeta \mathcal{A}[\![\mathsf{e_2}]\!]\ \mathbf{let\ x_2} = [\cdot]\ \mathbf{in}\ \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathbf{x_2} \\
&= \mathbf{let\ x_2} = [\cdot]\ \mathbf{in}\ \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathbf{x_2} \langle\!\langle \mathcal{A}[\![\mathsf{e_2}]\!] \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let\ x_2} = \mathcal{A}[\![\mathsf{e_2}]\!]\ \mathsf{in}\ \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathsf{x_2} && \text{by Theorem 5.8} \\
&\equiv_\zeta \mathcal{A}[\![\mathsf{e_1}]\!]\ \mathcal{A}[\![\mathsf{e_2}]\!]
\end{aligned}$$

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

Case: [Snd]

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

$\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x_1}] : (\mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A'.\, B']\!]) \Rightarrow \mathbf{B}$.

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

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

Goal (2) follows by Theorem 5.3 if we can show
(3) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{snd}\ \mathbf{x_1} : \mathcal{A}[\![B'[x := \mathsf{fst}\ e]]\!]$,
(4) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A'.\, B']\!] \vdash \mathcal{A}[\![\mathsf{snd}\ e]\!] \equiv \mathbf{snd}\ \mathbf{x_1}$, and
(5) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A'.\, B']\!] \vdash \mathcal{A}[\![\mathsf{snd}\ e]\!] : \mathcal{A}[\![B'[x := \mathsf{fst}\ e]]\!]$.

By $\rhd_\delta$, we are trying to show $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{snd}\ \mathcal{A}[\![e]\!] : \mathcal{A}[\![B'[x := \mathsf{fst}\ e]]\!]$. We have previously shown that $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'} \vdash \mathcal{A}[\![e]\!] : \mathbf{\Sigma\, x} : \mathcal{A}[\![A']\!].\, \mathcal{A}[\![B']\!]$. Using this fact with [Snd], we derive that
$\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma'}, \mathbf{x_1} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\Sigma\, x : A'.\, B']\!] \vdash \mathbf{snd}\ \mathcal{A}[\![e]\!] : \mathcal{A}[\![B']\!][\mathbf{x} := \mathbf{fst}\ \mathcal{A}[\![e]\!]]$.
We can derive our goal by [Conv] if we can show that $\mathcal{A}[\![B'[x := \mathsf{fst}\ e]]\!]$ is equivalent to $\mathcal{A}[\![B']\!][\mathbf{x} := \mathbf{fst}\ \mathcal{A}[\![e]\!]]$. By Theorem 6.5, we have that $\mathcal{A}[\![B'[x := \mathsf{fst}\ e]]\!]$ is equivalent to $\mathcal{A}[\![B']\!][\mathbf{x} := \mathcal{A}[\![\mathsf{fst}\ e]\!]]$. Focusing on $\mathcal{A}[\![\mathsf{fst}\ e]\!]$, we have:

$$
\begin{aligned}
&\mathcal{A}[\![\mathsf{fst}\ e]\!] \\
&\overset{\text{def}}{=} \mathcal{A}[\![e]\!]\ (\mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x_1}) \\
&= \mathbf{let}\ \mathbf{x_1} = [\cdot]\ \mathbf{in}\ \mathbf{fst}\ \mathbf{x_1} \langle\!\langle \mathcal{A}[\![e]\!] \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let}\ x_1 = \mathcal{A}[\![e]\!]\ \mathsf{in}\ \mathbf{fst}\ x_1 && \text{by Theorem 5.8} \\
&\rhd_\zeta\ \mathbf{fst}\ \mathcal{A}[\![e]\!]
\end{aligned}
$$

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

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

$$\mathcal{A}[\![\mathsf{snd}\ e]\!]$$

$$\begin{aligned}
&\overset{\text{def}}{=} \mathcal{A}[\![e]\!]\,(\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_1) \\
&= \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{snd}\ \mathbf{x}_1 \langle\!\langle \mathcal{A}[\![e]\!] \rangle\!\rangle && \text{by Theorem 6.4} \\
&\equiv \mathsf{let}\ \mathsf{x}_1 = \mathcal{A}[\![e]\!]\ \mathsf{in}\ \mathbf{snd}\ \mathsf{x}_1 && \text{by Theorem 5.8} \\
&\rhd_\zeta \mathbf{snd}\ \mathcal{A}[\![e]\!]
\end{aligned}$$

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

Case: [IF]

Must show

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

Let $\mathbf{K}_1 = \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![e_1]\!]\ \mathbf{K}\ \mathbf{else}\ \mathcal{A}[\![e_2]\!]\ \mathbf{K}$. Our conclusion follows by induction on $\Gamma \vdash e : \mathsf{Bool}$ if we show $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}' \vdash \mathbf{K}_1 : (\mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!]) \Rightarrow \mathbf{B}$. By [K-BIND], we must show (1) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}' \vdash \mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!]$ and (2) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!] \vdash \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}[\![e_1]\!]\ \mathbf{K}\ \mathbf{else}\ \mathcal{A}[\![e_2]\!]\ \mathbf{K} : \mathbf{B}$.

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

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

This follows by Theorem 5.3 if we can show
(3) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash \mathcal{A}[\![e_1]\!] : \mathcal{A}[\![\mathsf{B}'[\mathsf{x}' := e]]\!]$,
(4) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash \mathcal{A}[\![e_1]\!] \equiv \mathcal{A}[\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]$,
and
(5) $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash \mathcal{A}[\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] : \mathcal{A}[\![\mathsf{B}'[\mathsf{x}' := e]]\!]$.

Goal (3) follows from induction with the empty continuation and [CONV] if we can show that $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{x} \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\mathsf{Bool}]\!], \mathbf{p} : \mathbf{x} \equiv \mathbf{true} \vdash \mathcal{A}[\![\mathsf{B}'[\mathsf{x}' := e]]\!] \equiv \mathcal{A}[\![\mathsf{B}'[\mathsf{x}' := \mathsf{true}]]\!]$. By Theorem 6.5, we are trying to show $\mathcal{A}[\![\mathsf{B}']\!][\mathbf{x}' := \mathcal{A}[\![e]\!]] \equiv \mathcal{A}[\![\mathsf{B}']\!][\mathbf{x}' := \mathbf{true}]$. By [≡-REFLECT] and equivalence by $\rhd_\delta$, we can derive that under the extended environment $\mathcal{A}[\![e]\!] \equiv \mathbf{true}$, and we conclude with [≡-SUBST].

Focusing the right-hand side of goal (4), we have:

$$\begin{aligned}
&\mathcal{A}\,[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]\\
&\overset{\text{def}}{=} \mathcal{A}\,[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}\,[\![\mathsf{e}_2]\!])\\
&= \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\ \langle\!\langle \mathcal{A}\,[\![\mathsf{e}]\!]\rangle\!\rangle\\
&\text{by Theorem 6.4}\\
&\equiv \mathsf{let}\ \mathsf{x} = \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathsf{in}\ \mathbf{if}\ \mathsf{x}\ \mathbf{then}\ \mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\\
&\text{by Theorem 5.8}\\
&\rhd_\zeta\ \mathbf{if}\ \mathcal{A}\,[\![\mathsf{e}]\!]\ \mathbf{then}\ \mathcal{A}\,[\![\mathsf{e}_1]\!]\ \mathbf{else}\ \mathcal{A}\,[\![\mathsf{e}_2]\!]\\
&\equiv \mathcal{A}\,[\![\mathsf{e}_1]\!]\\
&\text{by } [\equiv\text{-}\textsc{If}\text{-}\eta_1] \text{ since } \mathcal{A}\,[\![\mathsf{e}]\!] \equiv \mathbf{true} \text{ by } [\equiv\text{-}\textsc{Reflect}]
\end{aligned}$$

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

Case: [ElimNat]

Must show

$$\begin{aligned}
&\mathcal{A}\,[\![\mathsf{e}]\!]\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\\
&\qquad \mathcal{A}\,[\![\mathsf{e}_1]\!]\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\\
&\qquad\qquad \mathcal{A}\,[\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\\
&\qquad\qquad\qquad \mathbf{K}[\mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2])))
\end{aligned}$$

has type $\mathbf{B}$. 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 $\mathbf{B}$, which can be proven by Theorem 5.2 if we show

$$\mathcal{A}\,[\![\mathsf{elimnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2]\!] \equiv \mathbf{elimnat}\ \mathcal{A}\,[\![\mathsf{A}]\!]\ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2$$

under the environment

$$\mathcal{A}\,[\![\Gamma]\!]\,, \boldsymbol{\Gamma}', \mathbf{y} \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e}]\!] : \mathcal{A}\,[\![\mathsf{Nat}]\!]\,, \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e}_1]\!] : \mathcal{A}\,[\![\mathsf{A}[\mathsf{zero} := \mathsf{x}]]\!]\,,$$
$$\mathbf{x}_2 \overset{\delta}{=} \mathcal{A}\,[\![\mathsf{e}_2]\!] : \mathcal{A}\,[\![\Pi\,\mathsf{n} : \mathsf{Nat}.\, \Pi\,\mathsf{x}' : \mathsf{A}[\mathsf{x} := \mathsf{n}].\, \mathsf{A}[\mathsf{x} := \mathsf{succ}\ \mathsf{n}]]\!]$$

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

$\mathcal{A}\llbracket \mathsf{elimnat}\ \mathsf{A}\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2 \rrbracket$

$\overset{\text{def}}{=} \mathcal{A}\llbracket \mathsf{e} \rrbracket\ (\mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket$
$(\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2)))$

$= \mathbf{let}\ \mathbf{y} = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket$
$(\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2))\langle\!\langle \mathcal{A}\llbracket \mathsf{e} \rrbracket \rangle\!\rangle$

by Theorem 6.4

$\equiv \mathsf{let}\ \mathsf{y} = \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathsf{in}\ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket$
$(\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathbf{y}\ \mathbf{x}_1\ \mathbf{x}_2))$

by Theorem 5.8

$\triangleright_\zeta\ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ (\mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathbf{x}_1\ \mathbf{x}_2))$

$= \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathbf{x}_1\ \mathbf{x}_2)\langle\!\langle \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket \rangle\!\rangle$

by Theorem 6.4

$\equiv \mathsf{let}\ \mathsf{x}_1 = \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ \ \mathsf{in}\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathbf{x}_1\ \mathbf{x}_2)$

by Theorem 5.8

$\triangleright_\zeta\ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ \ \mathbf{x}_2)$

$= \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ \ \mathbf{x}_2\langle\!\langle \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket \rangle\!\rangle$

by Theorem 6.4

$\equiv \mathsf{let}\ \mathsf{x}_2 = \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket\ \ \mathsf{in}\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ \ \mathbf{x}_2$

by Theorem 5.8

$\triangleright_\zeta\ \mathbf{elimnat}\ \mathcal{A}\llbracket \mathsf{A} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e}_1 \rrbracket\ \ \mathcal{A}\llbracket \mathsf{e}_2 \rrbracket$

□

Lemma A.2.

1. *If* $\vdash \Gamma$ *then* $\vdash \mathcal{A}\llbracket \Gamma \rrbracket$

2. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \mathbf{K} : (\mathcal{A}\llbracket \mathsf{e} \rrbracket : \mathcal{A}\llbracket \mathsf{A} \rrbracket) \Rightarrow \mathbf{B}$, *then* $\mathcal{A}\llbracket \Gamma \rrbracket, \mathbf{\Gamma}' \vdash \mathcal{A}\llbracket \mathsf{e} \rrbracket\ \mathbf{K} : \mathbf{B}$.

Proof. By mutual induction on $\vdash \Gamma$ and $\Gamma \vdash e : A$.

Case: [If] Let

$$\begin{aligned} K_1 = \textbf{let}\ x' = [\cdot]\ \textbf{in} \\ &\textbf{if}\ x' \\ &\textbf{then}\ \mathcal{A}[\![e_1]\!]\ (\textbf{let}\ x_1 = [\cdot]\ \textbf{in}\ (f\ x_1\ (\textbf{refl}\ x_1))) \\ &\textbf{else}\ \mathcal{A}[\![e_2]\!]\ (\textbf{let}\ x_2 = [\cdot]\ \textbf{in}\ (f\ x_2\ (\textbf{refl}\ x_2))) \end{aligned}$$

Then we must show

$$\begin{aligned} \mathcal{A}[\![\Gamma]\!], \Gamma' \vdash \textbf{let}\ f = \lambda\, y : \mathcal{A}[\![A[x := e]]\!]. \\ &\lambda\, p : y \equiv \mathcal{A}[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!].\ K[y] \\ &\textbf{in}\ \mathcal{A}[\![e]\!]\ K_1 : B \end{aligned}$$

According to [Let], we will first show that
$\mathcal{A}[\![\Gamma]\!], \Gamma' \vdash f : \Pi\, y : \mathcal{A}[\![A[x := e]]\!].\ \Pi\, p : y \equiv \mathcal{A}[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!].\ B$
By two applications of [Lam], we must show $\mathcal{A}[\![\Gamma]\!], \Gamma'.y : \mathcal{A}[\![A[x := e]]\!], p : y \equiv \mathcal{A}[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!] \vdash K[y] : B$. This follows by Theorem 5.2 if we can show $y \equiv N$. By [≡-Reflect], we have $y \equiv \mathcal{A}[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!]$. Combining this equivalence with our initial assumption using [≡-Trans], we achieve the goal.

Let $\textbf{ftype} = (\Pi\, y : \mathcal{A}[\![A[x := e]]\!].\ \Pi\, p : y \equiv \mathcal{A}[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!].\ B)$ and $\textbf{fbod} = (\lambda\, y : \mathcal{A}[\![A[x := e]]\!].\ \lambda\, p : y \equiv \mathcal{A}[\![\text{if } e \text{ then } e_1 \text{ else } e_2]\!].\ K[y])$. Next we will show $\mathcal{A}[\![\Gamma]\!], \Gamma', f \overset{\delta}{=} \textbf{fbod} : \textbf{ftype} \vdash \mathcal{A}[\![e]\!]\ K_1 : B$. We follow the same steps as the [If] case of Theorem A.1, but now must show

$\mathcal{A}[\![\Gamma]\!], \Gamma', f \overset{\delta}{=} \textbf{fbod} : \textbf{ftype}, x' \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\text{Bool}]\!], p' : x' \equiv \textbf{true} \vdash \mathcal{A}[\![e_1]\!]\ (\textbf{let}\ x_1 = [\cdot]\ \textbf{in}\ (f\ x_1\ (\textbf{refl}\ x_1))) : B[x := \textbf{true}]$.

This follows by induction on $\mathcal{A}[\![e_1]\!]$ if we can show $\textbf{let}\ x_1 = [\cdot]\ \textbf{in}\ (f\ x_1\ (\textbf{refl}\ x_1)) : (\mathcal{A}[\![e_1]\!] : \mathcal{A}[\![A[x := \text{true}]]\!]) \Rightarrow B[x := \textbf{true}]$. By [K-Bind], we must show

$\mathcal{A}[\![\Gamma]\!], \Gamma', f \overset{\delta}{=} \textbf{fbod} : \textbf{ftype}, x' \overset{\delta}{=} \mathcal{A}[\![e]\!] : \mathcal{A}[\![\text{Bool}]\!], p' : x' \equiv \textbf{true}, x_1 \overset{\delta}{=} \mathcal{A}[\![e_1]\!] : \mathcal{A}[\![A[x := \text{true}]]\!] \vdash f\ x_1\ (\textbf{refl}\ x_1) : B[x := \textbf{true}]$.

By applying [App], we are required to show $x_1 : \mathcal{A}[\![A[x := e]]\!]$. By the definition in the environment, we have that $\mathbf{x}_1$ has type $\mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{true}]]\!]$. By [Conv], we can show that $\mathbf{x}_1$ has type $\mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{e}]]\!]$ if we can show $\mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{e}]]\!] \equiv \mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{true}]]\!]$. By Theorem 6.5, this is the same as showing $\mathcal{A}[\![\mathsf{A}]\!][\mathbf{x} := \mathcal{A}[\![\mathsf{e}]\!]] \equiv \mathcal{A}[\![\mathsf{A}]\!][\mathbf{x} := \mathcal{A}[\![\mathsf{true}]\!]]$. By [$\equiv$-Subst], we must show that $\mathcal{A}[\![\mathsf{e}]\!] \equiv \mathcal{A}[\![\mathsf{true}]\!]$. By $\rhd_\delta$ we have $\mathbf{x}' \equiv \mathcal{A}[\![\mathsf{e}]\!]$, and by [$\equiv$-Reflect] we have $\mathbf{x}' \equiv \mathbf{true}$. By [$\equiv$-Symm] and [$\equiv$-Trans], we obtain our goal, since $\mathcal{A}[\![\mathsf{true}]\!] = \mathbf{true}$ by the ANF translation.

By applying [App] once more, we are required to show
$\mathbf{refl}\,\mathbf{x}_1 : \mathbf{x}_1 \equiv \mathcal{A}[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]$.

This can be shown by [Conv] if we can show $\mathcal{A}[\![\Gamma]\!], \mathbf{\Gamma}', \mathbf{f} \overset{\delta}{=} \mathbf{fbod} : \mathbf{ftype}, \mathbf{x}' \overset{\delta}{=} \mathcal{A}[\![\mathsf{e}]\!] : \mathcal{A}[\![\mathsf{Bool}]\!], \mathbf{p}' : \mathbf{x}' \equiv \mathbf{true}, \mathbf{x}_1 \overset{\delta}{=} \mathcal{A}[\![\mathsf{e}_1]\!] : \mathcal{A}[\![\mathsf{A}[\mathsf{x} := \mathsf{true}]]\!] \vdash (\mathbf{x}_1 \equiv \mathbf{x}_1) \equiv (\mathbf{x}_1 \equiv \mathcal{A}[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!])$. This can be shown by [$\equiv$-Cong-Equiv] if we can show $\mathbf{x}_1 \equiv \mathcal{A}[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]$. This is equivalent to showing $\mathcal{A}[\![\mathsf{e}_1]\!] \equiv \mathcal{A}[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]$, which is shown in the [If] case of Theorem A.1.

□