COMPILING WITH DEPENDENT TYPES

WILLIAM J. BOWMAN

Doctor of Philosophy
College of Computer and Information Science
Northeastern University

2018

Northeastern University
College of Computer and Information Science

PhD Thesis Approval

THESIS TITLE: Compiling with Dependent Types

AUTHOR: William J. Bowman

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

[illegible]	Nov 12, 2018.
Thesis Advisor	*Date*
[illegible]	Nov 12, 2018
Thesis Reader	*Date*
[illegible]	Nov. 12, 2018
Thesis Reader	*Date*
[illegible]	12 Nov 2018
Thesis Reader	*Date*
[illegible]	11/12/18
Thesis Reader	*Date*

GRADUATE SCHOOL APPROVAL:

[illegible]	11/29/18.
Director, Graduate School	*Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

[illegible]	11/29/18
Recipient's Signature	*Date*

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Dedicated to Daniel P. Friedman.

"No, I'm not going to grad school; I can't even imagine writing a dissertation."
— Me, responding to Dan

ABSTRACT

Dependently typed languages have proven useful for developing large-scale fully verified software, but we do not have any guarantees after compiling that verified software. A verified program written in a dependently typed language, such as Coq, can be type checked to ensure that the program meets its specification. Similarly, type checking prevents us from importing a library and violating the specification declared by its types. Unfortunately, we cannot perform either of these checks after compiling a dependently typed program, since all current implementations erase types before compiling the program. Instead, we must trust the compiler to not introduce errors into the verified code, and, after compilation, trust the programmer to never introduce errors by linking two incompatible program components. As a result, the compiled and linked program is *not verified*—we have no guarantees about what it will do.

In this dissertation, I develop a theory for preserving dependent types through compilation so that we can use type checking after compilation to check that no errors are introduced by the compiler or by linking. Type-preserving compilation is a well-known technique that has been used to design compilers for non-dependently typed languages, such as ML, that statically enforce safety and security guarantees in compiled code. But there are many open challenges in scaling type preservation to dependent types. The key problems are adapting syntactic type systems to interpret low-level representations of code, and breaking the complex mutually recursive structure of dependent type systems to make proving type preservation and compiler correctness feasible. In this dissertation, I explain the concepts required to scale type preservation to dependent types, present a proof architecture and language design that support type preservation, and prove type preservation and compiler correctness for four early-stage compiler translations of a realistic dependently typed calculus. These translations include an A-normal form (ANF), a continuation-passing style (CPS), an abstract closure conversion, and a parametric closure conversion translation.

ACKNOWLEDGMENTS

I never thought I would get a Ph.D. In fact, I thought I would never get a Ph.D. As I completed my undergraduate degree, a Ph.D. seemed like an insurmountable challenge: I would have to produce knowledge unknown to the world, write about, talk about it, and teach it. I didn't know how to do that. I didn't want to do that. I just wanted to write fun programs and learn interesting facts.

I was wrong. The following people helped me realize that.

- Dan Friedman

  Dan is the reason I'm here today.

  Dan introduced me to the beauty of programming languages, causing me to shift gears from software engineering and security to spending my free time reading about language design. When I was interested in research, Dan recommended I talk to Amal. He explicitly recommended that I not work with him, claiming that he doesn't do important research. That might be, but he does do important work.

- Amr Sabry

  Amr taught me how to write software, and gave me a few research problems to play with as a Master's student while I tried to figure out whether I wanted to do a Ph.D.

- R. Kent Dybvig

  Kent taught me that most programs aren't written by humans, and how to think like a machine.

- The Northeastern PRL, jointly and separately.

  Vincent St-Amour, Asumu Takikawa, Phillip Mates, Jamie Perconti, Paul Stansifer, Tony Garnock-Jones, Stephen Chang, Justin Slepak, Jonathan Schuster, Dionna Amalie Glaze, Andrew Cobb, Sam Caldwell, Ben Greenman, Leif Andersen, Daniel Patterson, Aaron Weiss, and Ben Lerner.

  Thank you all for hanging out at happy hours, criticizing my practice talks, and making my Ph.D. a great personal and professional experience.

- Max S. New

  Max, the category practitioner. He understands abstraction in ways I doubt I ever will. Many times, he has helped me understand my own work better. Max is a brilliant friend and amazing colleague.

- Cătălin Hriţcu

  Cătălin is a kind, generous, and skeptical person—a great scientist. He is one of the first people I met at a programming languages conference. Cătălin hosted me at Inria for a semester while I finished this work, giving me the chance to meet people who know way more about dependent types than me. This dissertation is much better because of that experience.

- Annabel Satin

  Annabel is an awesome person, working tirelessly in the background for the good of programming languages. Without her, our work might well grind to a halt.

- Youyou Cong

  Youyou visited us for a semester, and was critical to part of this dissertation. She has an encyclopedic knowledge of the literature surrounding CPS. I still sometimes bother her with my ideas.

- Neel Krishnaswami

  Every time I talk to Neel, I learn something awesome about semantics. Our chats have shaped my work in subtle ways.

- Ron Garcia

  Ron is the nicest person in the programming languages community. Talking with Ron, I feel like he is genuinely interested in what I have to say, even when all I have to say are half baked ideas. He can express his confusion, or clearly explain why I'm wrong, without making me feel stupid.

- Éric Tanter

  Éric has always made me feel like a colleague, and not just a graduate student. He's one of the reasons I'm sticking around in academia.

- Stephanie Weirich

  Stephanie has a unique perspective on dependent types. She has challenged my assumptions and design choices throughout my dissertation. One day, I'll try to put those thoughts in writing.

- Greg Morrisett

  Greg has been a huge help during this work. Even after starting as dean, Greg found time to chat about research. When I sent him a draft, he'd read it in mere hours and come back with cogent feedback, questions, and criticisms.

- Mitch Wand

  Mitch helped me learn more about writing and English during this dissertation than I learned in 22 years of school before I started.

- Matthias Felleisen

  Matthias has constantly pushed me to reach a higher standard for scientific rigor, precision, and clarity—more so than anyone else. He has never hesitated to (constructively) criticize my work. I've learned a ton from him, and if I'm lucky I'll learn some more.

- Amal Ahmed

  Amal has been a great advisor. She has absolute and infectious enthusiasm for what she does. She helped me learn about proofs and types, and about communicating complex mathematical work. She chewed me out when I ignored important things, and gave me latitude when I needed it.

  But mostly, Amal taught me to not fear hard problems. In 2010, as an undergrad, I asked Amal if she had any research problems. She gave me a book (Types and Programming Languages), and a secure compilation problem, and said it would take "6 months, max". In 2015, we finished that paper. In 2016, I had a choice: continue working on secure compilation, or switch to working on dependent types. I was more interested in dependent types, but switching topics 4 years in to a Ph.D. program seemed like a big risk, so I chose secure compilation. A few days later, I walked into her office and said "I changed my mind. It's going to be hard, but it'll be way more interesting.". I don't know if I could have done that without Amal.

Thank you all. You have all shaped this work, either directly or indirectly through emotional and psychological support, and are part of the reason I finished this dissertation. And thank you to many more people who I've not put in this list; there are many of you who have helped me in ways I haven't been able to put into words.

— William J. Bowman

SUPPORT

This work was funded by the National Science Foundation under grants NSF CCF-1203008, NSF CCF-1422133, and NSF CCF-1453796, and by the European Research Council under ERC Starting Grant SECOMP (715753) Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding agencies.

TYPOGRAPHICAL CONVENTIONS

I use a combination of colors and fonts to make distinctions between terms in different languages clear. Often, programs from more than one language will appear in the same equation. While which language each program belongs to will be clear from the structure of the equations, I use colors and fonts to make the distinction more apparent. These colors and fonts are chosen to be distinguishable in color, in grayscale, and with various forms of color blindness.

Each chapter will involve at least two languages: a source and a target language. Source languages expressions and metavariables are typeset in a blue, non-bold, sans-serif font. Target language expressions and metavariables are typeset in **bold, red, serif font**. These different fonts do not indicate *particular* languages—they only indicate source and target. I will remind the reader which language is which each time a new language is introduced.

For languages other than source and target languages, such as examples of concepts or from related work, I use a black, non-bold, serif font.

I use emphasis to introduce a word with a *technical definition*—*i.e.*, a specific technical use that may differ from the normal English definition.

Typographical Note. *I will remind the reader of these typographical conventions in the section I first use them, or when languages change, in a typographical note like this.*

ONE ROAD SHOULD NOT BE TAKEN

Two roads diverged in a blue and red wood,
A colorful forest of derivation trees.
I'm no traveler; I, therefore, could—
and, as a scientist, because I should—
followed each one into its leaves.

In one, I found the theory fair;
This I suggest; the other is worse.
In that, the types, one's mind ensnare;
Though it can work, I hereby declare.
Really, I beg you, just use the first.

CONTENTS

1 EXECUTING A VERIFIED PROGRAM

I begin this dissertation with a story; I will get technically precise starting in the next section.

In the beginning, computer programs were created. This was fine and was useful for doing math, and many programmers wrote many interesting programs. Sometime after, computers were created and began to execute computer programs. *This has made a lot of people very angry and been widely regarded as a bad move.*[1]

After computers were created, they became the primary readers and writers of computer programs. Some programs are written and read by humans, but most are written by *compilers* and read only by the computer. Hopefully, the compiler translates an input *source program* that is simple for a human to read and write into an *executable*—a, possibly optimized, program suitable for execution.

A computer will, faithfully and without question, execute the program it is given to execute. This is unfortunate because the program that the computer reads is almost never the one that the programmer intended to be executed. The resulting execution will surprise the programmer and enrage the user.

If we are very lucky, a bad execution will happen *despite* the best efforts of the programmer. Ideally, the programmer will have spent substantial effort trying to avoid letting a computer execute the wrong program. The programmer will have designed their program and reasoned about that design before ever writing a single line of code. Then, each line of code will have been written *carefully.* Surprisingly, we do occasionally get this lucky in practice.

Despite our luck, inevitably, the programmer will have been wrong. Probably, the design will have been flawed. Likely, the code will have had errors. So the computer will have faithfully executed an erroneous implementation of a flawed design.

Programming is fundamentally an act of communication and, as such, is susceptible to *miscommunication.* The programmer communicates, via their program, some process, or task, or other information. The program communicates this *very* precisely. So precisely, in fact, that it can be understood not only by other programmers, but by a "rock we tricked into thinking".[2] But, humans and rocks think very differently. The programmer often fails to understand how literally rocks think, and the rock fails to understand that the programmer doesn't always say what they *mean* what to say.

To avoid miscommunication, we can teach computers to *verify* their understanding of a program with the programmer through *program verification*—separately specifying what the programmer meant for the program to do, and then, before executing the program, making the computer verify that the program does what the programmer meant for it do. A *verified program* is a program implementation with a specification

Technically, verification only checks that the program does what we said *we meant for it to do, so miscommunication is still possible. For this dissertation, I assume this is good enough.*

1 From Adams (1980).
2 @daisyowl. `https://twitter.com/daisyowl/status/841802094361235456`. March 14 2017.

and a proof that the implementation meets the specification. By verifying a program, we reduce miscommunication and thus reduce program errors, by providing high assurance that the programmer and the computer are "on the same page"—the programmer has written what they expect to happen, then written the program, and the computer has agreed that the program will indeed do what is expected.

Program verification is useful for reducing program errors but, thanks to compilers, essentially no verified program has ever been executed. I make this statement more precise later in Section 1.3, but for now, here is what I mean. Recall that compilers, not programmers, write most programs, and this is particularly true of executables. To produce a verified program, a compiler would have to transform the program implementation, its specification, and its proof, which is beyond the scope of current compiler design. Even if the compiler were proven to satisfy some compiler correctness specification and could compile a verified source program, the compiler would not produce a verified target program. That is, we could not get an executable implementation complete with an executable-level specification that corresponds to the original specification and an independently checkable proof that the executable meets its specification. Contemporary verified compilers only transform the implementation. We may think that composing the proof of correctness for the compiler with the proof of correctness for the program yields a proof that the generated executable is verified, but this (typically) only tells us that the executable reduces to the same value as the original verified program, not that the executable meets the original specification. Evaluation to the same value is a weak specification: it requires the compiler only operates on *whole programs* and does not capture (for example) security properties that guarantee secret values aren't leaked to attackers. Thus even when a programmer has gone through the trouble of verifying a program, the program that actually executes is an *unverified* executable.

Although, if it was compiled with a verified compiler, it is probably a correct unverified executable.

In this dissertation, I theorize a solution to the problem of compilers transforming verified programs into unverified executables. Before I can precisely describe this theory, I must precisely describe program verification and the problems that arise when attempting to compile programs, verified or not.

1.1 Program Verification and Dependent Types

The key elements of program verification are:

1. a *specification*, which describes the intended behavior or properties of a program and is independent of the program's implementation,
2. an *implementation*, which describes in code how to accomplish the goal of the program (this is what we typically think of as a *program*),
3. a *proof*, which witnesses the fact that the implementation meets its specification.

There are many ways to instantiate the above elements into a system for program verification, and additional criteria we wish to consider for a given instantiation. For example, we want specifications to be simple to understand and write, and expressive enough to capture full functional, safety, and security properties. We also want proofs to be easy to check, and ideally for the proof system to be *logically consistent*—*i.e.*, the existence of a proof should imply that the implementation does actually meet the specification.

A particularly common and useful (as discussed shortly) way to implement program verification is by using *dependent types* (in particular, *full-spectrum* dependent types)[3]. In the terminology of typed programming, *dependent types* allow for *types*, the language for statically describing program behaviors, to *depend on* (*i.e.*, to include as part of the language of types) some or all *terms*, the language for dynamically implementing programs. This is in contrast to traditional typed programming language, such as Java or ML, in which there is a *phase distinction* between types and terms, meaning types exist at compile-time and *cannot* refer to terms, which exist later at run-time. This restricts what specifications can be expressed in types. *Full-spectrum* dependent types refers to the ability of types to depend on *arbitrary* terms, instead of only part of the language of terms. In that case, there is no syntactic distinction between types and terms—together, they make up a single language of *expressions*.

Henceforth, I use the unqualified dependent types to mean full-spectrum dependent types.

As an example of program verification with dependent types, consider attempting to verified the following operations on lists from a standard typed language such as ML.

```
length : List A -> Nat
length = ...

append : List A -> List A -> List A
append = ...
```

In this example, we declare that the term `length` has the type `List A -> Nat`, meaning that it is a function that takes a `List A` as input and produces a `Nat` as output. In a type system with phase distinction, `List A`, `->`, and `Nat` are all drawn from the language of types, while `length` and `append` are in the language of terms.

In a full-spectrum dependently typed language, we could write the above example with the more precise type below thus using type checking to verified a correctness property.

```
length : List A -> Nat
length = ...

append : (l1 : List A)
         -> (l2 : List A)
         -> {l : List A | length l == length l1 + length l2}
append = ...
```

3 These were once called *fully dependent types* (Xi, 1998), but contemporary work uses term *full-spectrum* (Weirich et al., 2017).

In this example, we name each argument to `append` when describing its type. The first argument is named `l1`, and the second is `l2`. Then, we can refer to that argument, the term, in the type. We can also write other arbitrary terms in the types. We use this to declare that the output of `append` is a `List A`, named `l`, such that the `length` of `l` is equal to the sum of the `length`s of `l1` and `l2`. This means that not only does `append` return a `List A`, but a `List A` of a particular `length`, capturing a correctness specification in the types.

Using dependent types, we can build a program verification system that satisfies all desired criteria listed earlier. We write specifications using types (in particular, dependent types), and write implementations in the terms of the dependently typed language. The proofs are represented by the annotations and additional terms in the implementation itself and are checked via type checking; for example, the implementation of a function may include an additional argument whose type represents a precondition, and the term passed in for that argument represents a proof that the precondition holds if the term is well-typed.

1.1.1 The Virtues of Dependent Types

DEPENDENT TYPES SIMPLIFY SPECIFICATION BY AVOIDING SPECIFICATION PUNS Dependent types simplify reading and writing specifications and increase assurance that the specification describes the actual implementation by avoiding *specification puns*.

In languages without full-spectrum dependent types, such as those with indexed types or refinement types, types cannot directly refine all terms. Instead, types can only depend on a subset of terms or a separate language of type-level computations. This prevents types from directly refining the behavior of some terms and introduces a gap between the specification and the implementation. Instead, either the rest of the behavior is unspecified or a separate specification-level model of the implementation must be constructed and the implementation proven to correspond to this model. This gap can introduce *specification puns*, meaning a specification refers to something that is *like* the implementation but is technically defined differently—*i.e.*, a specification that exploits two possible meanings of what looks like the same expression.

For example, in a language with refinement types and phase distinction in which types may depend on (and only on) first-order data, we cannot directly prove the same correctness property for `append` that we saw earlier. Instead, we write the example from earlier as follows.

```
length : (l : List A) -> Nat
length = ...

length' := ...

append : (l1 : List A)
           -> (l2 : List A)
```

```
          -> {l : List A | length' l == (length' l1) + (length' l2)}
append = ...
```

Unlike before, we cannot refer to `length`, a term, directly in the type of `append`. Instead, we must write a separate, type-level, version of `length`—called `length'` in this example, for clarity—and use a separate, type-level, version of `+`. This is a pun; really, `length'` has a different definition than `length`, even though a sufficiently clever language might let us name them both `length` by maintaining the names in two different namespaces. Similarly, the `+` used in the type is not the run-time addition function, although anyone who reads the specification might reasonably assume it is.

Pragmatically speaking, specification puns create two problems. First, a programmer must write any definition that is used in a specification twice, and pass these two representations around *everywhere* that specification is used. Instead of simply using the term `length` to specify the behavior of `append`, we must duplicate `length` in the types. Second, without doing additional work, we are less confident that the run-time program is correct, since we can only prove correctness of type-level copies. For example, we do not know that the run-time function `length` corresponds to the type-level version `length'`, and so it is unclear that `append` actually returns a list whose `length` is the expected value, as opposed to one whose `length'` is the expected value. We would first have to prove that `length` and `length'` coincide, which is difficult in the general case.

This problem of specification puns arises in program verification using *program logics* and *logical frameworks*. In program verification based on these frameworks, the specification and proof languages are separate from the implementation language. This is an advantage in one sense—there is more freedom in how programs can be implemented vs how they are specified and proven—but a disadvantage since it introduces puns.

For example, Proof-Carrying Code (*PCC*) (Necula, 1997) has been used to address the problem of generating verified executables discussed earlier, but relies on a logical framework and runs into the problems of puns. Necula (1997) uses the Edinburgh Logical Framework (*LF*) to encode specifications and proofs about assembly code, and develops a compiler that can produce verified programs for programs whose specifications are expressed and provable in first-order logic. However, LF used this way requires a phase distinction, so specifications and proofs cannot refer to assembly directly, and there are puns. For example, addition either refers to assembly addition instruction `ADD` or the LF constant `+`. To prove that the assembly code `ADD r1,r2,r1`—which shuffles bits in registers—produces the value `l` in `r1`, the logic actually asks that we prove `r1 + r2 = l`—which represents mathematical addition on integers. For this proof about `r1 + r2 = l` to represent anything about the assembly instruction `ADD`, we must ensure some correspondence between the LF constants and the assembly code, thus creating additional work and making it less clear that specifications correspond to implementations. In the case of our `append` example, to verify our implementation of `append` from above, we first add the LF constant `length'`, then demonstrate that `length'` in the logic corresponds to `length`, then prove that `append` is correct.

With full-spectrum dependent types, we avoid the problem of puns by directly using any term in any type, decreasing programmer effort and increasing confidence. As seen earlier, the type of `append` can be written using full-spectrum dependent types as follows.

```
append : (l1 : List A)
           -> (l2 : List A)
           -> {l : List A | length l = (length l1) + (length l2)}
```

This requires no separate type-level definition `length'`, and instead use the run-time function `length` directly in the specification of `append`.

The expressivity of dependent types depends on which axioms are admitted in the particular language, which I discuss further in the next chapter.

DEPENDENT TYPES SIMPLIFY PROOF THROUGH ABSTRACTION Dependent types are extremely expressive as a specification and proof system. The earlier example was a simple programming example, but dependent type systems are also able to express sophisticated mathematical theorems and frameworks, such as category theory (Gross et al., 2014; Timany and Jacobs, 2015). This means program verification using dependent types can easily leverage abstract mathematical concepts and proofs. They have also been used to encode program logics (Nanevski et al., 2008; Krebbers et al., 2017), thus allowing the programmer to embed alternative program verification systems if they so desire. In short, dependent types can express whatever specifications and proofs we want, and by starting from dependent types we capture a range of program verification systems instead of just one.

Usually, this gain in expressivity of the logic comes at a cost of expressivity as a programming language. For example, dependently typed languages such as Coq and Agda disallow expressing arbitrary loops—every term must be provably terminating, so they cannot express arbitrary general recursive functions. This is because non-termination can easily introduce inconsistency, since the usual typing rules for recursive functions will give an infinite loop any type and thus introduce a proof of any specification, or break decidability of type checking, since checking a type that contains a term may require partially evaluating that term causing the type checker itself to diverge on infinite loops. However, it is possible to regain non-termination while maintaining logical consistency or decidability by separating possibly diverging programs from proofs via a modality (Jia et al., 2010; Casinghino et al., 2014; Nanevski et al., 2006; Nanevski and Morrisett, 2005; Swamy et al., 2013).

The extra expressivity can also come at the cost of automation. For example, Necula (1997) restricts specifications to those expressed in first-order logic. This is a loss for expressivity, but a gain for automation and simplifies the design of the compiler: the compiler can generate proofs automatically, rather than requiring the user to write them and designing the compiler to transform and preserve them.

1.2 Executing a Dependently Typed Program

A dependently typed program has never been executed. Only machine code is ever executed, and so far there is no dependently typed machine code.

Instead, dependently typed programs are first type checked, then compiled (sometimes called *extracted*) to some executable. For example, Coq[4] programs are typically compiled to OCaml, or Haskell, or sometimes Scheme. During this process, all types are erased, even when the target is OCaml or Haskell, which are typically considered typed languages. This means that after compiling dependently typed programs, we no longer have the specification or a machine-checkable proof of correctness for the executable.

In Coq, we could prove some data invariant, use that to safely write optimized functions, then compile to OCaml. For example, we could write a Coq function that returns the head of a list without checking whether the list is empty or not, as long as the `length` is statically guaranteed to be non-zero.

```
Inductive List (A : Set) : Nat -> Set :=
| nil : List A 0
| cons : forall (h:A) (n:Nat), List A n -> List A (1 + n).

Definition head {A} {n : Nat} (v : List A (1 + n)) :=
  match v with
  | cons _  h _ t => h
  end.

(* Example invalid := head nil. *)
Example valid : (Nat -> Nat) := head (cons _ (fun x => x) 0 (nil _)).
```

This declares `List` to be an inductively defined datatype, parameterized by some element type `A` and indexed by a `Nat`, which statically represents the length of the `List` in the type. There are two constructors for `List`s: `nil`, which produces a `List` of `A`s of length `0`, and `cons`, which expects an element `h` and a list of length `n` and produces a `List` of length `1 + n`. We then define `head` to take a non-empty `List`, *i.e.*, a term `v` of type `List A (1 + n)`, and simply return the `h` field of the `cons`. Note that we don't need to consider the case for when `v` is `nil`, since the type guarantees that can never happen. We can create and type check a simple example by calling `head` on a `List` of length `1` containing the function `fun x => x`.

After compiling `head` to OCaml, we have the following function.

```
let head _ = function
| Nil -> Obj.magic __
| Cons (h, _, _) -> Obj.magic h
```

4 https://coq.inria.fr/

$$\begin{tikzcd}
\text{Coq} \arrow[d, "\text{Compile}"'] & \\
\text{OCaml} \arrow[d, "\text{Compile}"'] & \text{Driver} \arrow[l, harpoon, "\text{Link}"] \\
\text{x86} & \text{FFI} \arrow[l, harpoon', "\text{Link}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{Coq} \arrow[d, "\text{Miscompilation Errors}"'] & \\
\text{OCaml} \arrow[d, "\text{Miscompilation Errors}"'] & \text{Driver} \arrow[l, harpoon, "\text{Linking Errors}"] \\
\text{x86} & \text{FFI} \arrow[l, harpoon', "\text{Linking Errors}"']
\end{tikzcd}$$

Figure 1.1: Transforming a Verified Coq Program into an Unverified Executable

```
(** val valid : nat0 -> nat0 **)

let valid =
```

First, notice that the code is littered with `Obj.magic`, an unsafe cast operation, and has none of the original types. Essentially all static checking from OCaml's type system is turned off. What was originally a well-typed verified program is now an unverified, and in fact, unsafe program. Next, notice that this OCaml implementation behaves differently than the implementation in Coq. In Coq, we could prove that `head` only received non-empty lists, and could omit the null check. In OCaml, this is no longer true and the compiler inserts a case for `Nil` whose behavior is mysterious.

In a typical Coq workflow, we would next link this OCaml program with other unverified components, such as some program driver—a top-level function to initialize the main loop or a user interface—then compile the program to assembly using OCaml's compiler, and possibly link with low-level code such as cryptography libraries or run-time systems. The left side of Figure 1.1 shows a diagram of this workflow. Unfortunately, as seen on the right side of Figure 1.1, this workflow leads to producing an unverified executable from the verified Coq program by introducing miscompilation errors and linking errors. In this diagram, the arrows represent compilation, while the harpoons represent linking.

Miscompilation errors are errors introduced into a program as a result of a program error in the compiler. For example, suppose a compiler is designed to perform the following optimization, reordering conditional branches to maximize straight-line code.

$$\texttt{if e then e}_1\texttt{ else e}_2 \rightarrow \texttt{if (not e) then e}_2\texttt{ else e}_1$$

However, if the compiler is implemented incorrectly, it may instead perform the following incorrect transformation, introducing a miscompilation error.

$$\texttt{if e then e}_1\texttt{ else e}_2 \rightarrow \texttt{if (not e) then e}_1\texttt{ else e}_2$$

The conditional is negated, but the branches are not switched. Now, when $\texttt{e}_1$ should have executed, $\texttt{e}_2$ will, and vice versa, even if the original program was verified.

Real miscompilation errors are unlikely to be so simple to understand, and there are lots of them (Yang et al., 2011). For example, at the time of writing, there is 1 open compilation error in the Coq to OCaml compiler[5], and at least 3 open miscompilation errors in the OCaml compiler[6].

Miscompilation errors can be greatly reduced by *compiler verification*—applying program verification to the compiler itself by proving that the compiler always produces programs that equate, in some way, to the input program. This is the approach taken by projects like the CompCert C compilers (Leroy, 2009), the Cake ML compiler (Kumar et al., 2014), and the CertiCoq compiler (Anand et al., 2017). Of course, the compiler itself is written and verified in a high-level language and then compiled. At the very least, this means we can never execute the verified compiler, and at worst, leaves us vulnerable to "trusting trust" attacks (Thompson, 1984). But, this has been shown effective at reducing miscompilation errors in practice; only two miscompilation errors have been found in the verified CompCert C compiler, compared to the hundreds in other compilers (Yang et al., 2011). And these two errors were caused by the unverified driver linked into the verified code.

Standard compiler verification does not rule out a more insidious problem—*linking errors*—as demonstrated by the two errors in the verified CompCert compiler. *Linking errors* are errors introduced when linking two components of a program when one component violates some specification of the other or of the over all program. This can happen when linking two verified components if we never check that the specifications are compatible, or when linking an unverified component with a verified one. Trying to describe errors in CompCert as an example is too complex, so instead, consider the function `head`, introduced earlier, which has a precondition expressed in its type that it is only called with a non-empty `List`. If we ever link it with a component that calls `head` with an empty list, this would be a linking error: the resulting execution would be undefined.

In a strongly typed source language, linking errors should not be possible, but linking typically happens *after* compiling and compiler target languages are not strongly typed. As we saw in the above example, after compiling from Coq to OCaml, the compiled version of the function `head` is essentially untyped. This is also true of the assembly produced by the OCaml compiler.

As a result, it is easy to introduce linking errors in executables, even when produced from verified programs. Recall from Figure 1.1 that it is standard practice to compile Coq code to OCaml, then link with some program driver. We can introduce a linking error by linking with one line of code in OCaml using the verified `List` functions from Coq. First we add another verified Coq function, `applyer`, which applies the first element of a `List` of functions to the first element of another `List`.

```
Definition applyer {n : Nat} (fs : List (Nat -> Nat) (1 + n))
                             (ns : List Nat (1 + n)) :=
  (head fs) (head ns).
```

5 https://github.com/coq/coq/issues/7017. Accessed Aug 3 2018.
6 https://caml.inria.fr/mantis/view_all_bug_page.php. Accessed Dec 3 2018. Filtered for major, crash, or blocking unresolved open issues, and manually inspected for miscompilation bugs.

This is compiled to the following OCaml code. As in the Coq program, this simply applies the first element of the list of functions to the first element of the list of natural numbers. Notice that it uses `Obj.magic` to essentially disable type checking, since extracted code may not be well-typed in general.

```
(** val applyer : nat0 -> (nat0 -> nat0) list -> nat0 list -> nat0 **)

let applyer n fs ns =
  Obj.magic head n fs (head n ns)
```

Next, we link with the following line of code in OCaml. This line exists in a separate module from `applyer` and simply tries to call `applyer` on two non-empty lists. The programmer who wrote this module may be relying on the type checker to detect errors, and may not know that the module they are using has used `Obj.magic`.

```
let _ = applyer 1 (Cons (0, O, Nil)) (Cons ((fun x -> S x), O, Nil))
```

If the reader cares for a brief diversion, there are three other type errors; I give the answers at the end of the next section.

This line could almost be mistaken for a well-typed use of `applyer`. It calls the function with a `list` of numbers and a `list` of functions on numbers. Unfortunately, the arguments are in the wrong order, so OCaml attempts to call the number `0` as a function, and crashes.

This simple error does not even take advantage of OCaml's inability to check dependent types. We could, in principle, ascribe OCaml types to `applyer`, and rule this linking error out. However, in general, it is easy to construct examples that rely on dependent types, higher-order invariants, or freedom from state and non-termination (since Coq is purely functional). Even a fully annotated OCaml program with no uses of `Obj.magic` cannot check these, and it is easy to violate each when linking in OCaml. Furthermore, eventually we compile the program to assembly, which has no type system, and it is likely that when we link in assembly (with low-level libraries, run-time environments, drivers, etc) we can easily cause linking errors there too.

The simple example we've just seen shows that, after all the trouble of verifying a program, the state-of-the-art gives the programmer few guarantees. They have thrown away the help of the type checker, must manually reason about all the invariants when linking, and have no systematic way to check that the compiler has correctly implemented the verified program. This example function is a relatively simple program; C compilers, operating systems, cryptographic primitives and protocols, and other high-assurance software developed in dependently typed languages are not simple, and we cannot expect a programmer to manually check all the invariants after compiling and linking these programs. This is a problem: the end result of program verification, after the compilation and linking, is an *unverified program*. What good is the effort spent on program verification under these conditions?!

$$\begin{tikzcd}
\text{Coq} \arrow[d, "\text{Compile w/ CertiCoq}"'] & \\
\text{Clight} \arrow[d, "\text{Compile w/ CompCert}"'] & \text{Driver} \arrow[l, "\text{Linking Errors}"] \\
\text{x86} & \text{FFI} \arrow[l, "\text{Linking Errors}"']
\end{tikzcd}$$

Figure 1.2: Verified Compilation of a Verified Coq Program into an Unverified Executable

$$\begin{tikzcd}
\text{Coq} \arrow[d, "\text{Compile w/ CertiCoq}"'] & \text{Coq} \arrow[d] & \\
\text{Clight} \arrow[d, "\text{Compile w/ SepCompCert}"'] & \text{Clight} \arrow[l, "\text{Safe Linking}"] \arrow[d] & \text{Driver} \arrow[ll, bend right, "\text{Linking Errors}"'] \\
\text{x86} & \text{x86} \arrow[l, "\text{Safe Linking}"] & \text{FFI} \arrow[ll, bend left, "\text{Linking Errors}"]
\end{tikzcd}$$

Figure 1.3: Verified Separate Compilation of Verified Coq Programs into an Unverified Executable

1.3 Preserving the Verified-Ness of a Program

Admittedly, there is much work on developing verified compilers and this work is being extended to dependently typed languages. A verified compiler is proven to be free of miscompilation errors, typically stated as a refinement: if the source program evaluates in the source semantics to a value, then the compiled program executes to a related value in the target semantics. For example, the CompCert C compiler mentioned earlier is a verified compiler for a subset of C called *Clight* (Leroy, 2009). The CakeML compiler is a verified compiler for ML (Kumar et al., 2014). Work is ongoing to develop CertiCoq, a verified compiler for Coq (Anand et al., 2017). By combining the compiler correctness theorem with the correctness of the original Coq program, CertiCoq supports preserving the verified-ness of the source program—as long as we never link with anything. We end up with the guarantees described in Figure 1.2. We use verified compilers to assembly to rule out miscompilation errors. However, this still leaves the possibility of linking errors.

A compiler that guarantees correctness of separate compilation, what I'll call a *separate compiler*, is verified to be free of linking errors as long as all components being linked are compiled with the same verified compiler. Separate compilers exist for non-dependently typed languages, such as SepCompCert, an extension of the CompCert to support separate compilation (Kang et al., 2016). CertiCoq has support for separate compilation, so it can provide the guarantees in Figure 1.3. As long as all components are compiled, separately, linking between those components is guaranteed to be safe, and

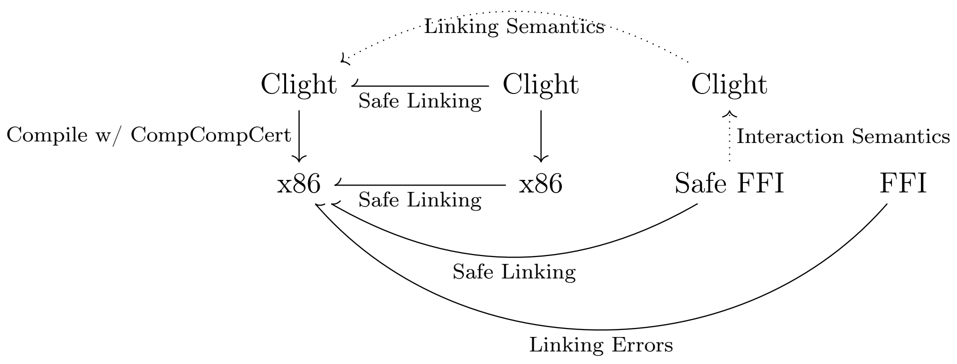

Figure 1.4: Verified Compositional Compilation into an Unverified Executable

the resulting program is still verified in the following sense: by composing the original proof of correctness and proof of correctness of separate compilation, we have a proof that the executable will behave according to the original specification when linked with other verified components that have been compiled with the verified separate compiler. Unfortunately, there is no way to *automatically check* that linking will be safe, although we will know it will be true. There may still be unverified drivers or low-level libraries we need to link with, so linking errors are still possible, and we have no way for the *automatically check* for them, *i.e.*, for the machine to detect and report potential linking errors prior to linking.

A *compositional compiler* excludes further linking errors by specifying what target language components are safe to link with independently of the compiler used to produce those components, if any. For example, CompCompCert allows linking between Clight programs and x86 programs, as long as there is a safe *interaction semantics* specifying their interoperability (Stewart et al., 2015). Pilsner allows safe interoperability between an ML-like language component and an assembly component as long as there exists a *PILS relation* between the assembly and some ML component, and the interoperability between the two components at the ML-level is safe (Neis et al., 2015). The details of these cross-language relations is not important; the point is the kinds of linking they support. These both lead to the guarantees like those in Figure 1.4, in which it is possibly to safely interact with components that were not verified and then compiled with a verified compiler. In this diagram, the dotted arrows represent the relation specifying interaction between assembly components and Clight components so that the programmer can reason about linking with assembly in terms of some Clight behaviors. However, because there is no general way to *automatically check* the safety of the assembly components—*i.e.*, to check for the existence of a *PILS relation* or *interaction semantics* specification—these too still admit some linking errors. There is no existing work on a compositional compiler for dependently typed languages.

They key problem with the above separate and compositional compilers is not in their guarantees—it's fine to restrict what we can safely link with—but in *automatically detecting* possible linking errors. All information about what is it safe to link with is

$$\begin{tikzcd}
\text{Coq} \arrow[d, "\text{MythicalComp}"'] & & \\
\text{Annotated x86} \arrow[d, "\text{Erase Annotations}"'] & \text{Annotated x86} \arrow[l, "\text{Safe Linking}"'] & \text{x86} \arrow[ll, dotted, bend left, "\text{Disallowed: possible linking error}"] \\
\text{x86} & &
\end{tikzcd}$$

Figure 1.5: Verified Compilation into a Verified Executable

in the *theorems* about the compilers, and none of that information is in the program or components generated by the compiler, *i.e.*, in the executable. This requires the programmers to strictly follow the workflow and check the provenance of all components before linking in order to consider the resulting executable verified. That is, part of the proof that the executable is verified cannot be checked by a machine—it must be checked and enforce by the programmer. In my view, calling the resulting executable verified is only true with a trivial definition of the word—the program is verified if the programmer can, after spending enough effort and without any quickly checkable artifact, convince themselves that they have taken a verified program and not introduced any miscompilation or linking errors.

This is the technical sense in which I meant "essentially no verified program has ever been executed"

To call an executable verified, we need a way to describe in the language of the executable what linking is permitted, what is not, and give a procedure by which a machine can automatically verify that no miscompilation or linking errors have been introduced during compilation and linking. Only then have we preserved the verified-ness in a meaningful way, and only then could we ever execute a verified program. Diagrammatically, this mythical compiler would support the guarantees seen in Figure 1.5. This would allow compiling and linking with arbitrary correctly annotated assembly without giving a specification of that assembly in terms of some other languages. The annotations would be used to prove the absence of linking errors. We could compile into an annotated assembly, check that the assembly still meets its specification, then use that specification to automatically detect safe linking and possible linking errors. After linking, then, and only then, could we erase the annotations to generate machine code, and use a standard verified compiler to prove that the machine code is still verified. Since there is no more linking, there is no more risk of linking errors at the machine-code-level.

We can preserve the verified-ness of a verified program developed in a dependently typed language by designing a *type-preserving* compiler for dependently typed languages, *i.e.*, by preserving dependent types and well-typedness through compilation. I discuss type preservation further in Chapter 3, but for now the idea is this: by preserving all the dependent types through compilation into the target language of the compiler, all specifications are preserved and can be used at link time to rule out linking with code that violates the specification. Since well-typed programs also represent proofs, the compiler also preserves the proof through compilation. We can then check that the proof is still valid according to the preserved specification, verifying that no miscompilation errors were introduced that violate the specification. Since the specification is still around at the low-level, we can use type checking to enforce that specification on external components before linking. The end result of compiling a verified program in a dependently typed language with a type-preserving compiler is still a verified program, in a meaningful sense of the word verified—the machine can automatically check the whole proof, and detect and rule out linking errors.

And all we have to do is develop a compiler that can automatically translate dependent types into equivalent dependent types describing low-level code (eventually, assembly code), transform programs in such a way that the program can be executed on a low-level machine (eventually, on hardware), and transform proofs about high-level functional code into proofs about low-level assembly code that can be automatically and decidably checked.

To see how preserving types detects and prevents errors, consider the miscompilation error from earlier. Suppose we start with the following conditional expression, but this time, we give it a specification using dependent types.

```
e : if x then A else B
e = if x then e1 else e2
```

Now the expression `e` has a dependent type: when `x` is `true`, then the type is `A`, otherwise the type is `B`. If a miscompilation error occurs as before, then we get the following expression.

```
e+ = if (not x) then e1 else e2
```

But now, we can detect the miscompilation via type checking. We expect that `e+ : if x then A else B`, but instead find that `e+ : if (not x) then A else B`. As these types are not equal, type checking reveals something has gone wrong with compilation.

Note that if the source specification is weak, then this does not necessarily guarantee freedom from miscompilation errors, so we must still verify the compiler to rule out all miscompilation errors. But it turns out that merely by preserving dependent types, we are almost forced to build a proven *correct* compiler, *i.e.*, a compiler that guarantees that programs run to the equivalent values before and after compilation; I discuss this in Chapter 3.

The big benefit of type preservation is that linking errors are easily ruled out via type checking. In the earlier example, linking with the following line of code caused our verified program to crash.

```
let _ = applyer 1 (Cons (0, O, Nil)) (Cons ((fun x -> S x), O, Nil))
```

By preserving types into the target language and, importantly, ensuring the target type system expresses the same invariants as the source, we would be forced to show that this is well-typed with respect to the original type:

```
applyer : forall n : Nat, List (Nat -> Nat) (1 + n) ->
                             List Nat (1 + N) -> Nat
```

Just as we would see in Coq, the type system would automatically report the various type errors: that `1` and `0` are OCaml `int`s, not the compiled form of Coq `Nat`s; the first `List` contain functions not `Nat`s; and vice versa for the second `List`.

With dependent types, type preservation offers us a final benefit: proof-carrying code is possible for essentially arbitrarily expressive specifications. Types (*i.e.*, specifications) are preserved through the compiler into the executable. With dependent types, the types can express essentially specifications—full-functional correctness, safety, and security properties. Well-typedness (*i.e.*, proofs that implementations satisfy specifications) is also preserved. At the end of a type preserving compiler, we have the implementation with its specification and can independently check the proof.

With a verified dependent-type-preserving compiler from Coq to dependently typed assembly, we could live the dream—we could use type checking to rule out linking errors, link the components into a whole program, type check the program to have a machine-check proof that the specification still hold independent of any trust in a compiler, linker, or implementation, and—as a final step—erase types and run the program. The program that executes would be *guaranteed* to be the one the programmer intended to execute!

Barring incorrect or under specification, hardware bugs, gamma rays, errors in physics, or a vindictive deity.

1.4 Thesis

In this dissertation, I demonstrate that, in theory, we can design the mythical compiler just described for a dependently typed language such as Coq, and I describe that theory. My thesis is:

> Type-preserving compilation of dependent types is a theoretically viable technique for eliminating miscompilation errors and linking errors.

This thesis is only a first step toward the grand story described earlier in this chapter.

I demonstrate this thesis by studying four translations which model the front-end of a type-preserving compiler for a realistic dependently typed calculus. These translations are the A-normal form (ANF) translation, continuation-passing style (CPS) translation, and two variants of closure conversion. The aim is to support all the core features of dependency found in Coq and preserve them through these standard compiler translations. I pick Coq as the “ideal” dependently typed language due to the significant number of contemporary verified programs developed using Coq, the accessibility of examples of linking errors, and the availability of a simple core calculus (the Calculus of Inductive Constructions, *CIC*) corresponding to its core language. In the final chapter, Chapter 8, I review these translations, summarize the lessons, and explain some of the remaining open questions.

In this dissertation, I do not go beyond demonstrating that type preservation is *theoretically* viable. I leave many practical questions unaddressed to focus on the theory

$$\begin{tikzcd}
ECC^D \arrow[d, swap, "\text{Chapter 4 (ANF)}"] & CoC^D \arrow[d, swap, "\text{Chapter 6 (CPS)}"] \arrow[dd, bend left, "\text{Chapter 7 (Parametric CC)}"] \\
ECC^A \arrow[d, swap, "\text{Chapter 5 (Abstract CC)}"] & CoC^k \\
ECC^{CC} & CoC^{CC}
\end{tikzcd}$$

Figure 1.6: Compilers in this Dissertation

of preserving dependent types. By the end of this dissertation, I will not be able to compile all of Coq, but I will be able to support all the core dependent-type features found in CIC. I will also not be able to target assembly, but I will show a design that scales to realistic dependently typed functional languages, such as Coq, and show this design works with common compiler transformations for functional languages.

1.5 Contributions of This Dissertation

The compiler translations I develop are described in Figure 1.6. On the left are translations that scale to all the core features of dependency found in Coq. I present these first as they should scale to Coq in their current form, and they are simpler to understand, although I developed these last based on lessons learned and problems exposed when developing the translations on the right. The core calculi used for these translations are based of the Extended Calculus of Constructions (ECC) (Luo, 1990). On the right are extensions of the translations that, historically, have been studied and used for type preservation. The core calculi for these translation are based on the Calculus of Constructions (Coquand and Huet, 1988). In their current form, these translations do not scale to some features of dependency.

Below, I summarize the contributions in this dissertation and explain how this dissertation is organized. I also include an appendix for each language and translation with complete definitions.

ESSENCE OF DEPENDENT TYPES In Chapter 2, I introduce dependent types formally. I start by discussing the key features of dependency, *i.e.*, what makes dependent types *dependent*. Then I present a calculus, ECC^D, that is used as the source language for the translations in Chapter 4 and Chapter 5. ECC^D includes the core features of dependency found in Coq, although it omits features inessential to dependency, such as recursive functions.

TYPE-PRESERVING COMPILATION In Chapter 3, I introduce type-preserving compilation and compiler correctness formally. I start with a brief history of type-preserving compilation. I then discuss the key difficulties in developing a type-preserving compiler for a dependently typed language, and present the proof architecture used throughout this paper. I also discuss type-preserving compilation as a technique for semantics modeling, which I use to prove logical consistency of my dependently typed target languages.

A-NORMAL FORM In Chapter 4, I give a type-preserving A-normal form (ANF) translation from ECC^D to ECC^A, an ANF-restricted variant of ECC^D with a machine-like evaluation semantics. The ANF translation is used to make control flow explicit by naming intermediate computations. I show how ANF can fail to be type preserving with dependent types, and how to recover type preservation. I discuss the relation to the well-known failure of type preservation for CPS.

ABSTRACT CLOSURE CONVERSION In Chapter 5, I give a type-preserving abstract closure conversion from ECC^D to ECC^{CC}. Since ECC^A is a restriction of ECC^D, the translation also applies to ECC^A, and I show that the closure conversion preserves ANF. Closure conversion (CC) is commonly used in functional languages to make first-class functions simple to allocate in memory, but this variant of type-preserving closure conversion is not commonly used. I show why the commonly used closure conversion translations do not scale to some features of dependency, whereas abstract closure conversion does.

This chapter was previously published as Bowman and Ahmed (2018). Compared to the published version, this chapter is extended to include dependent conditionals (discussed in Chapter 2), and a proof that ANF is preserved through the abstract closure conversion. It also includes a correction to the structure of the proof of one lemma; the theorems are unaffected.

CONTINUATION-PASSING STYLE In Chapter 6, I give a type-preserving Continuation-Passing Style (CPS) translation from CoC^D to CoC^k. I start by restricting ECC^D to CoC^D, which excludes a key feature of contemporary dependently typed languages (higher universes). I discuss historical problems preserving dependent types through CPS translation, and how we can overcome these problems in some settings. I develop a type-preserving CPS translation for CoC^D and prove compiler correctness, but also show that translation is not type preserving for ECC^D. In Chapter 8, I conjecture how this translation could be extended to translate all of ECC^D.

This chapter was previously published as Bowman et al. (2018). The chapter is essentially the same as the published version, but includes a correction to the structure of the proof of one lemma; the theorems are unaffected.

PARAMETRIC CLOSURE CONVERSION In Chapter 7, I give a type-preserving parametric closure conversion translation from CoC^D to CoC^{CC}. I extend commonly used closure conversion translations based on *existential types* to dependent types. I show that this translation is type preserving and prove compiler correctness for CoC^D, but demonstrate that this translation is not type preserving for ECC^D. In Chapter 8, I conjecture how this translation could be extended to translate all of ECC^D.

CONCLUSIONS In Chapter 8, I summarize the lessons of the four translations presented in this dissertation. I conjecture how these lessons extend to the final two compiler translations necessary for a prototype type-preserving compiler to assembly, and how the CPS and parametric closure conversion target languages might be extended to support the missing features of dependency. I end by speculating about future work—problems that will need to be addressed to make that compiler not merely theoretically viable, but practical.

2 ESSENCE OF DEPENDENT TYPES

In this chapter, I formally introduce full-spectrum dependent types. I am not making any advances in type theory; all the features I present are well-known and commonly used in contemporary dependently typed languages. Therefore, I avoid any detailed discussion of the meta-theory or mathematical semantics of dependent types and focus on the interpretation and use of these features as a programming language and program verification system.

I start by introducing the key features of *dependency*, that is, features whose expressions e haves types that refer to a sub-expression of e. These are the features that distinguish dependently typed languages from standard typed languages, and are all found in contemporary dependently typed languages such as Coq, Agda, Idris, and F*.

I then build a dependently typed source calculus, ECC^D, with each of these features. ECC^D is the basis for the source language used in the rest of this dissertation; I use it as the source language for Chapter 4 and Chapter 5, although, I remove one key feature from ECC^D starting in Chapter 6. Recall that I use Coq as my "ideal" source language as its core language is relatively close to a small core calculus and it is used for many large-scale program verification projects. ECC^D is close to Coq in terms of the features of dependency, although it is missing two pragmatic features—recursive functions and computational relevance—that are orthogonal to dependency, which I discuss in Chapter 8. If we can develop a type-preserving compiler for ECC^D, it ought to be possible to scale type preservation to Coq.

2.1 Essential Features of Dependency

Typographical Note. *In this section, I use a* black, non-bold, serif font *to typeset examples of semantic rules and expressions for a dependently typed language that is never completely defined.*

The following features represent the core type theoretic structure found in contemporary dependently typed languages. If we must restrict or omit any of these features to build a type-preserving translation, then it cannot scale to Coq without further work.

Each feature below consists of some typing rules, some run-time reduction rules used to run programs, and some equivalence principles used to decide equivalence between types while type-checking. Equivalence should be sound with respect to reduction, *i.e.*, if e reduces to e', then e is also equivalent to e'. This is what allows reduction to take place in the type system, and thus how types can compute and refine terms. I omit explicit equivalence rules when I have given the reduction rules.

2.1.1 Higher Universes

Recall that in full-spectrum dependent types, there is no syntactic distinction between terms and types—there are only expressions—so we need to describe the type of a type. *Universes* are the types of types, and *higher universes* make it possible to write specifications about types, or specifications about specifications about types, and so on. This is useful for formal mathematics and generic programming, both of which are commonly done in dependently typed languages.

The typing rules for universes are given below.

$$\frac{\vdash \Gamma}{\Gamma \vdash \text{Prop} : \text{Type}_0} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}}$$

This presentation uses the usual presentation of universes a-la Russell, where we have no syntactic distinction between what can appear on the left- or right-hand side of the colon (:). There is one base universe, Prop, the type of types or type of *propositions*. Prop itself has a type, Type_0. I use the meta-variable U to refer to either Prop or some higher universe Type_i.

When source programmers can write down universes other than a single base universe, *i.e.*, when the programmer can write down Type_0 or Type_i and not just Prop, the type system has higher universes. By disallowing the programmer from writing any universe other than the base universe, we can get away with exactly two universes: Prop, which the programmer can use, and its type Type_0, which only the type system can use when type checking Prop. In this case, Prop is often written as $\star$ and Type_0 is written $\Box$. With higher universes, each universe Type_i also needs a type, Type_{i+1}.

Some languages, such as Coq and F*, use multiple base universes with different interpretations. For example, Coq uses two base universes, Set and Prop, where (loosely speaking) terms whose types have type Set are interpreted as *computationally relevant* (*i.e.*, as program implementations), while terms whose types have type Prop are interpreted as *computationally irrelevant* (*i.e.*, as proofs with no run-time behavior). I ignore this kind of distinction for this dissertation.

Cumulativity is a pragmatic feature related to higher universes found in some dependently typed languages. It allows a type in a lower universe to also implicitly inhabit a higher universe; it is a form of subtyping for universes. Coq supports cumulativity, while Agda does not. With cumulativity, the following typing rules are admissible, although they are usually derived from a subtyping judgment.

$$\frac{\vdash \Gamma}{\Gamma \vdash \text{Prop} : \text{Type}_i} \qquad \frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \text{Type}_i : \text{Type}_j}$$

Universes support no reduction rules or equivalence principles.

Digression. *Universes are used in the type system as a type-level witness to being a type. Recall that we have no syntactic distinction between terms and types, but universes* *can give us a way of classifying terms and types. An expression e of type U (either* Prop *or* Type_i*) is a type, whereas an expression e of type A such that A is not a universe is a term.*

Despite this, types may still be required at run-time, contrary to our intuition about terms and types. Deciding whether an expression is computationally relevant is still an active area of research (Tejiŝĉák and Brady, 2015). For example, Coq provides a distinction between its two base universes, so deciding whether expressions in those universes are relevant is simple. But for higher universes, a sound and approximate static analysis is used. Work on explicit relevance annotations does not have a complete story for arbitrary inductively defined data or recursive functions (Mishra-Linger, 2008).

2.1.2 Dependent Functions

Dependent functions give the programmer the ability to write functions whose result type refers by name to the arguments of the function. This allows encoding pre and postconditions on the function and representing theorems with universal quantification.

The typing rules for dependent functions are given below.

$$\frac{\Gamma \vdash A : \text{Type}_i \qquad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Pi\, x : A.\, B : \text{Type}_i} \qquad\qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B}$$

$$\frac{\Gamma \vdash e : \Pi\, x : A'.\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\ e' : B[e'/x]}$$

As with all types in a dependent type system, types themselves have types. Therefore, we have a typing rule for the type of dependent functions $\Pi\, x{:}A.\, B$, whose type is a universe Type_i. Dependent function types are introduced by the usual anonymous function $\lambda\, x : A.\, e$ and eliminated through application $e\ e'$. In the typing rule for application, we see dependency—the sub-expression e' is copied into the type system to instantiate x. This copying is the key problem in type preserving compilation, essentially because we need to transform both e''s interpretation as a program for compilation and e''s interpretation as a type-level expression for type-checking.

Dependent functions support the usual run-time operation, β-reduction, defined below.

$$\Gamma \vdash (\lambda\, x : A.\, e_1)\ e_2 \quad \rhd_\beta \quad e_1[e_2/x]$$

It is undesirable to allow η-reduction in the presence of other common features in dependently typed languages, as it breaks *subject reduction* (preservation of types under reduction).

$$\lambda\, x : A.\, \text{f}\ x \not\rhd_\eta \text{f}$$

For example[1], if we combine the η-reduction with cumulativity, then subject reduction does not hold in the system. Suppose $\lambda\, x : A.\, \mathrm{f}\ x : \Pi\, x : \mathrm{Type}_1.\, \mathrm{Type}_1$ but $\mathrm{f} : \Pi\, x : \mathrm{Type}_2.\, \mathrm{Type}_1$ due to cumulativity. After η-reduction, the type changes to a super-type of the original term, hence subject reduction does not hold. As such, η-reduction is excluded from Coq, which supports cumulativity. Agda, which does not have cumulativity, supports η-reduction, but the feature is being reconsidered as it breaks subject reduction when combined with dependent record types.[2]

While η-reduction is problematic, we can safely include η-equivalence for dependent functions. Adding η-equivalence directly, instead of deriving it from η-reduction, avoids the problems of combining η-principles with cumulativity. We can include η-equivalence by adding the following (somewhat informal) equivalence rule.

$$\frac{\text{when } e \text{ is a function}}{\Gamma \vdash e \equiv \lambda\, x : e\ x.}$$

There are different ways to implement this equivalence, which primarily differ in how to decide whether e is a function. The simplest technique is to make equivalence type-directed and use the type of e to decide. Coq uses an untyped equivalence, which I choose to adopt and present in the next section.

η-equivalence is included in some, but not all, dependently typed languages. Older version of Coq omitted and current versions of F* exclude it, but recent versions of Coq, Idris, and Agda include it.

Digression. *Unfortunately, by adding η-equivalence instead of deriving it from η-reduction, we (necessarily) complicate the judgmental structure of the dependent type system. It means equivalence is not derived simply from the reduction, but is an auxiliary judgment. Each new judgment complicates the type system and any reasoning about the type system, and means the choice of how the equivalence is defined will affect type preservation. However, disrupting subject reduction is at least as problematic, since a compiler might want to reduce terms,* e.g.*, to perform optimizations. I discuss how the choice of equivalence judgment affects type preservation further in Chapter 3.*

There is an additional concern for typing dependent function when we have higher universes—*predicativity*, *i.e.*, what universe the type of the function argument can have. The earlier typing rule for dependent function types is *predicative* because it requires that the argument type have the same universe as the result type, *i.e.*, the function is quantifying over expressions that "live" in a universe no higher ("larger") than the universe of expressions the function produces. Dependent functions can also support *impredicative* quantification, in which the function quantifies over expressions

1 This example is reproduced from the Coq reference manual, `https://coq.inria.fr/distrib/current/refman/language/cic.html`.

2 `https://github.com/agda/agda/issues/2732`

in a *higher* universe than the expressions the function produces. A common way to support impredicativity is to add the typing rule below.

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \qquad \Gamma, x : A \vdash B : \mathrm{Prop}}{\Gamma \vdash \Pi\, x : A.\, B : \mathrm{Prop}}$$

Impredicative dependent functions strictly increases the expressivity of the type system. For example, with impredicativity we can encode the polymorphic identity function as $\mathrm{f} : \Pi\, \alpha : \mathrm{Prop}\, .\, \alpha \to \alpha$ and apply it to itself $(\mathrm{f}\ (\Pi\, \alpha : \mathrm{Prop}\, .\, \alpha \to \alpha)\ \mathrm{f}) : \Pi\, \alpha : \mathrm{Prop}\, .\, \alpha \to \alpha$. Without impredicativity, we could not write this example.

However, impredicativity can easily lead to inconsistency. Martin-Löf's original presentation of type theory allowed arbitrary impredicative quantification (Martin-Löf, 1971) by declaring $\mathrm{Prop} : \mathrm{Prop}$, but is inconsistent (Girard, 1972) (which I understand via Coquand (1986), as I'm not sufficiently fluent in French to read the original). We can also get inconsistency by allowing impredicativity at more than one universe in the hierarchy (Girard, 1972) (again, via Coquand (1986)); note that the above impredicative rule allows impredicative quantification only when the dependent function is in the universe Prop. Impredicativity is also inconsistent when combined with some axioms, such as set-theoretic excluded middle and computational relevance.

The above tension between additional expressivity and inconsistency makes impredicativity a contentious feature. Some dependently typed language allow it, some reject it entirely, and some meet somewhere in between. Current versions of Agda reject impredicativity, making it impossible to build certain models such as an embedding of impredicative System F in Agda's base universe Set. By default, Coq allows impredicativity in Prop but not Set, to support set-theoretic reasoning about programs in Set which must be computationally relevant. Dependent Haskell (Weirich et al., 2017), a dependently typed variant of Haskell, allows arbitrary impredicativity and gives up logical consistency in favor of a more expressive but still type safe language.[3]

Digression. *Dependent function types are also called* Π types*,* dependent products*,* indexed Cartesian products*, or* Cartesian products of a family of types*. The latter names suggesting products seem to be commonly used in mathematical settings, and are related to how dependent functions are given mathematical semantics. As product is suggestive of pairs, I exclusively use dependent functions to suggest their interpretation as functions.*

To give an intuition about their nature as pairs, we can view dependent functions $\Pi\, x : A.\, B$ *as an infinite* lazy *pair* $B[x_0/x] \wedge B[x_1/x] \wedge ...B[x_i/x] \wedge \ldots$*, for all* $x_i : A$.

3 While Dependent Haskell is a realistic contemporary example, the idea of admitting inconsistency in favor of other pragmatic considerations was seriously considered much earlier by Cardelli (1986).

The finite representation is a function. To see this concretely, we can encode a finite lazy pair using dependent functions as follows.

$$\begin{array}{lcl} A \times B & = & \Pi\, x : \text{bool}.\, \text{if}\, x\, \text{then}\, A\, \text{else}\, B \\ \text{fst}\, e & = & e\ \text{true} \\ \text{snd}\, e & = & e\ \text{false} \\ \langle e_1, e_2 \rangle & = & \lambda\, x : \text{bool}.\, \text{if}\, x\, \text{then}\, e_1\, \text{else}\, e_2 \end{array}$$

This requires booleans and dependent elimination of booleans, discussed later in Section 2.1.4. The type of pairs $A \times B$ is defined as a dependent function that, when given a boolean, returns is something of type A or B depending on the value of the boolean argument. We provide true *to get a A and* false *to get B. This yields a 2-element pair, since there are two booleans. We can imagine that if the dependent function was quantifying over a larger type, with infinitely many elements, how this could represent an infinite pair.*

2.1.3 Dependent Pairs

Dependent pairs allows programmers to write a pair in which the type of the second component refers to the first component by name. This allows encoding an expression e paired with a proof e that satisfies some specification, and representing theorems with existential quantification.

The typing rules for dependent pairs are given below.

$$\frac{\Gamma \vdash A : \text{Type}_i \qquad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma\, x : A.\, B : \text{Type}_i} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash \langle e_1, e_2 \rangle\, \text{as}\, (\Sigma\, x : A.\, B) : \Sigma\, x : A.\, B}$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \text{fst}\, e : A} \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \text{snd}\, e : B[\text{fst}\, e/x]}$$

Dependent pair types have the type Type_i of each component. Unlike dependent functions, it is always inconsistent to allow impredicativity for dependent pairs (Girard, 1972; Coquand, 1986). The introduction rule is a pair $\langle e_1, e_2 \rangle$ as $\Sigma\, x : A.\, B$ of components e_1, of type A, and e_2 of type $B[e_1/x]$. Note that since we cannot, in general, decide how to abstract the type of e_2 from the particular e_1, pairs must be annotated. For brevity, we omit the type annotation on dependent pairs, as in $\langle e_1, e_2 \rangle$. The elimination form for dependent pairs are first $\text{fst}\, e$ and second $\text{snd}\, e$ projections of dependent pairs. Like in the typing rule for application, in the typing rule for second projection $\text{snd}\, e$, we see dependency—the first component of the pair e_1 can be referred to by the name x in the type B of the second component.

The reduction rules for dependent pairs are the usual projection rules for pairs, presented below.

$$\begin{array}{rcl} \Gamma \vdash \text{fst}\,\langle e_1, e_2\rangle & \rhd_{\pi_1} & e_1 \\ \Gamma \vdash \text{snd}\,\langle e_1, e_2\rangle & \rhd_{\pi_2} & e_2 \end{array}$$

In this dissertation, I do not require additional equivalence rules, in particular, an η-equivalence rule. η-equivalence is defined for dependent pairs in Agda (more generally, for record types, which are iterated dependent pairs), but not in Coq. As I did not find a need for η-equivalence of dependent pairs, I choose to follow Coq and omit it.

One may expect that we could encode dependent pairs using dependent functions, similar to how we Church-encode pairs using functions, and thus expect dependent pairs are not an essential feature. Such an encoding is not possible in general; for example, it is not possible in the impredicative Calculus of Constructions (Streicher, 1989), and thus not possible in Coq. Therefore, we need to consider dependent pairs as a separate and distinct feature of dependency.

Digression. *Dependent pairs are also called* dependent sums*,* strong Σ types*,* indexed disjoint union*,* disjoint unions of families of types*, and (extremely confusingly, since it is also used to refer to dependent functions)* dependent products*. As with dependent functions, I choose to focus on their interpretation as pairs and use dependent pairs exclusively.*

We can view a dependent pair $\Sigma\, x : A.\, B$ as an infinite sum $B[x_0/x] \vee B[x_1/x] \vee \ldots B[x_i/x] \ldots$ for all $x_i : B$. To get an intuition for this, we can recover finite sums using dependent pairs and booleans in a dual construction to dependent functions.

Encoding finite sums using dependent pairs also requires η-equivalence for booleans, but I ignore this for simplicity.

$$\begin{array}{lcl} A + B & = & \Sigma\, x : \text{bool}.\, \text{if}\, x\, \text{then}\, A\, \text{else}\, B \\ \text{inj}_1\, e & = & \langle \text{true}, e\rangle \\ \text{inj}_2\, e & = & \langle \text{false}, e\rangle \\ \text{case}\, e\, \text{of}\, x_1.\, e_1; x_2.\, e_2 & = & \text{if fst}\, e\, \text{then}\, e_1[\text{snd}\, e/x_1]\, \text{else}\, e_2[\text{snd}\, e/x_2] \end{array}$$

Strongly related to dependent pairs are *existential types*, a restriction of dependent pairs that allows elimination only by pattern matching and where the first component can never be depended upon in elimination. In fact, dependent pairs are sometimes called *strong dependent pairs* while existential types are called *weak dependent pairs*; I exclusively used existential type or existential package. The typing rules for existential types are below.

$$\frac{\Gamma \vdash A : \text{Type}_i \qquad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \exists\, x : A.\, B : \text{Type}_i} \qquad \frac{\Gamma \vdash A : \text{Type}_i \qquad \Gamma, x : A \vdash B : \text{Prop}}{\Gamma \vdash \exists\, x : A.\, B : \text{Prop}}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e' : B[e/x]}{\Gamma \vdash \text{pack}\,\langle e, e'\rangle\, \text{as}\, \exists\, x : A.\, B : \exists\, x : A.\, B}$$

$$\frac{\Gamma \vdash e : \exists\, x : A.\, B \qquad \Gamma, \alpha : A, x : B \vdash e' : C \qquad \Gamma \vdash C : \text{Type}_i}{\Gamma \vdash \text{unpack}\,\langle \alpha, x\rangle = e\, \text{in}\, e' : C}$$

The introduction form is still a pair, although it is distinguished from dependent pairs by the pack keyword. However, the elimination for requires pattern matching the type C produced by pattern matching does not contain any reference to x or α. This prevents us from defining the second projection operation $\mathrm{snd}\, e$ when the type of the second component of e depends on the first component. This restriction in the elimination rule makes impredicative existential types consistent.

Existential types are not an essential feature for dependency since they can be defined using dependent functions (Coquand, 1986). However, the ability to support impredicativity makes them useful for some encodings and compiler transformations, as I discuss further in Chapter 5 and Chapter 7.

2.1.4 Dependent Conditional

Dependent conditional refers to dependent elimination of a sum type, such as booleans or (non-dependent) sums $A + B$. The key features are: (1) that the elimination form is branching and (2) that the type of each branch can be different, depending on the value e being branched on. This allows programming with branches that depend on the value of the conditional and encoding theorems with (finite) disjunction.

I choose to use booleans with a dependent if. The key typing rules are below.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathrm{bool} : \mathrm{Prop}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{true} : \mathrm{bool}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{false} : \mathrm{bool}}$$

$$\frac{\Gamma, \mathrm{y} : \mathrm{bool} \vdash B : U \qquad \Gamma \vdash e : \mathrm{bool} \qquad \Gamma \vdash e_1 : B[\mathrm{true}/\mathrm{y}] \qquad \Gamma \vdash e_2 : B[\mathrm{false}/\mathrm{y}]}{\Gamma \vdash \mathrm{if}\, e\, \mathrm{then}\, e_1\, \mathrm{else}\, e_2 : B[e/\mathrm{y}]}$$

The introduction forms are the usual true and false, but notice that the elimination form is dependent. When branching on an expression e, the type of each branch B can be different depending on the value of e.

Digression. *Some dependent conditional construct is necessary, although which particular construct doesn't matter as each can be encoded with the other. We use the same construction to encode finite sums using booleans and dependent pairs as presented in the previous section, and encode booleans using finite sums as follows.*

$$\begin{array}{lcl} \mathrm{bool} & = & 1 + 1 \\ \mathrm{true} & = & \mathrm{inj}_1\, \langle\rangle \\ \mathrm{false} & = & \mathrm{inj}_2\, \langle\rangle \\ \mathrm{if}\, e\, \mathrm{then}\, e_1\, \mathrm{else}\, e_2 & = & \mathrm{case}\, e\, \mathrm{of}\, \_ . \, e_1 ; \_ . \, e_2 \end{array}$$

Digression. *The dependent conditional is the only feature of dependency commonly presented as* positive type*—essentially, a type whose constructors are considered primary and whose elimination form is simply pattern matching on the constructors. This is*

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

Figure 2.1: ECC^D Syntax

interesting, as it also introduces the most difficulty for type preservation, as I discuss in Chapter 4 and Chapter 6.

It is unclear if this is because of the presentation as a positive type. The presentation as a negative type requires a sequent calculus presentation where sequents support multiple conclusions. This is not done in any dependently typed language, however, like type preservation for dependent types, combining sequent calculus and dependent types has been a long open problem and is still ongoing work (Miquey, 2018).

2.1.5 What About Inductive Types

Those familiar with dependent types may be surprised at the omission of a common and seemingly critical feature from the feature set just presented: inductive types. Most dependently typed languages—including Coq, Agda, Idris, and F*—feature primitive inductively defined datatypes, which allow the programmer to add new datatypes over which terminating functions can be defined. This both allows the programmer to add new datatypes and simplifies logical consistency by providing a schematic way of generating well-founded inductive structures. However, we can instead encode inductive datatypes using the following features of dependency and recursive types. For example, Altenkirch et al. (2010) give a dependently typed calculus in which inductive datatypes are encoded using dependent functions, dependent pairs, dependent conditionals (all introduced below), and recursive types. (This idea should be familiar from the simply typed and even untyped settings.) Since recursive types introduce no new expressiveness in terms of dependency, I consider them inessential.

2.2 A Representative Source Calculus

In this section I present ECC^D, a source calculus with all the core features of dependency described in the previous section. I include a complete reference in Appendix A, with any elided portions of figures. I start by presenting the judgments related to type checking, then move on to judgments related to evaluation and compilation.

Typographical Note. *In this section, I typeset ECC^D as a source language, that is, in a* blue, non-bold, sans-serif font, *because it will serve as a source calculus in later chapters.*

ECC^D is meant to capture the features of dependency found in contemporary dependently typed languages, such as Coq, without any features orthogonal to dependency and type preserving compilation. This allows us to study type preservation and dependency without worrying about extraneous details.

ECC^D is also designed to simplify type preservation as much as possible while still being a convincing representation of a realistic dependently typed language. I add one feature that is not essential to dependency but simplifies some proofs—definitions, which I introduce shortly. I also carefully design the judgmental structure of ECC^D. In particular, I choose to implement non-type-directed equivalence through (non-type-directed) reduction. This complicates η-equivalence slightly, but greatly simplifies type preservation as I discuss in Chapter 3. The choice is also justified—the Calculus of Inductive Constructions, on which Coq is based, uses the exact same design.

2.2.1 Type System

The language ECC^D is Luo's Extended Calculus of Constructions (*ECC*) (Luo, 1990) extended with definitions (Severi and Poll, 1994) and dependent conditionals (booleans and dependent if). ECC itself extends the Calculus of Constructions (*CoC*) (Coquand and Huet, 1988) with dependent pairs and higher universes.

I present the syntax of ECC^D in Figure 2.1. As described in Chapter 1, there is no explicit phase distinction, *i.e.*, there is no syntactic distinction between terms and types; there are only expressions. For clarity, I will usually use the meta-variable e to evoke a term, and the meta-variables A and B to evoke a type. For variables, I usually use Latin letters such as x and y to evoke a term variables, and Greek letters such as α and β to evoke a type variables. Of course, since there is no formal distinction, the above conventions will break down. The expressions include all the features presented in the previous section, plus let expressions $\mathsf{let\,x = e\,in\,e'}$, which are used to introduce definitions. Note that let-bound definitions in ECC^D do not include type annotations; this is not standard, but type checking is still decidable, and it simplifies compilation (Severi and Poll, 1994). I discuss how this affects the translation in Chapter 4.

All expressions are typed and reduced with respect to a local environment Γ. Environments contain assumptions or axioms $\mathsf{x} : \mathsf{A}$, which declares that the name x has type A or, equivalently, that x is a proof of A, and definitions $\mathsf{x} = \mathsf{e}$, which declares that the name x is defined to be the expression e. For simplicity, I typically ignore the details of capture-avoiding substitution. This is standard practice, but is worth pointing out explicitly anyway.

In Figure 2.2, I define the *reduction* relation $\Gamma \vdash \mathsf{e} \rhd \mathsf{e'}$, *i.e.*, a single step of evaluation. This reduction relation is used both for running programs and deciding equivalence between types during type checking, therefore reduction is defined over open terms.

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

$$
\begin{array}{rll}
\Gamma \vdash (\lambda\, x : A.\, e_1)\; e_2 & \rhd_\beta & e_1[e_2/x] \\
\Gamma \vdash \mathsf{fst}\,\langle e_1, e_2\rangle & \rhd_{\pi_1} & e_1 \\
\Gamma \vdash \mathsf{snd}\,\langle e_1, e_2\rangle & \rhd_{\pi_2} & e_2 \\
\Gamma \vdash \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_1} & e_1 \\
\Gamma \vdash \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_1} & e_2 \\
\Gamma \vdash x & \rhd_\delta & e \qquad \text{where } x = e \in \Gamma \\
\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' & \rhd_\zeta & e[e'/x]
\end{array}
$$

Figure 2.2: ECC^D Reduction

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

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

$$
\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x = e' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rhd^* \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}\ \textsc{Red-Cong-Let}
\qquad \ldots
$$

Figure 2.3: ECC^D Conversion (excerpts)

While the reduction relation is untyped, *i.e.*, not type directed, it is only guaranteed to be terminating on well-typed terms. The reduction relation include all the reduction rules presented previously in Section 2.1, plus the two reduction rules for definitions: (1) δ-reduction of a variable to its definition from the current local environment and (2) ζ-reduction of a let-expression by substitution. As we will see in the typing rules for let, definitions are introduced during type checking, not reduction, thus δ-reduction and ζ-reduction are not redundant.

In Figure 2.3, I give the main rules for the conversion relation $\Gamma \vdash e \rhd^* e'$. The remaining congruence rules are entirely uninteresting; nevertheless, I define them in Figure A.3. *Conversion* is the reflexive, transitive, congruence closure of the reduction relation—it applies any reduction rule, any number of times, under any context. Intuitively, conversion is used to normalize terms in types and to normalize type-level computation, and thus decide type equivalence. It tells us when one type *converts* to another. It also serves to define the for ECC^D.

The only surprising conversion rule is Rule Red-Cong-Let, which allows a definition to be introduced when applying conversion to a let expression. This means there are two, confluent, ways to evaluate a let expression: $\Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, e' \rhd_\zeta e'[e/x]$, or

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x = e \vdash x \rhd_\delta e}{\vdots} \qquad \dfrac{\dots}{\vdots} \qquad \dfrac{\Gamma, x = e \vdash x \rhd_\delta e}{\vdots}}{\Gamma, x = e \vdash e' \rhd^* e'[e/x]}}{\Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, e' \rhd^* \mathsf{let}\, x = e \,\mathsf{in}\, e'[e/x]} \qquad \dfrac{}{\Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, e' \rhd_\zeta e'[e/x]}}{\Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, e' \rhd^* e'[e/x]}$$

The advantage is that *definitions*, essentially a delayed substitution that is introduced during type checking, allow δ-reduction during type-checking. In a non-dependent setting, we could just write an application anywhere we might write let, since the two programs reduce to exactly the same thing. However, with definitions, δ-reduction essentially allows what would be a β-reduction at run-time to happen during type-checking, providing an additional type equivalence. For example, the β-reduction $(\lambda\, x : A.\, e')\; e \rhd_\beta e'[e/x]$ provide (syntactically, through substitution) the knowledge that $x \equiv e$, but only after the function is type checked and allowed to evaluate. Instead in the let expression above, definitions record that $x = e$ while type checking the body e', and provide the equivalence $x \equiv e$ via δ-reduction. This is useful for type preservation, as I show in Chapter 4 and Chapter 5.

Digression. *Definitions aren't necessary to get this additional equivalence. The same effect can be achieved by simulating definitions using the identity type to provide an explicit proof of equality* $x = e$ *via an additional argument to a function. Then, by eliminating this in the body of the function, the body is type-checked under the additional equivalence. That is, we can define* $\mathsf{let}\, x = e \,\mathsf{in}\, e'$ *using the identity type as follows.*

$$\mathsf{let}\, x = e \,\mathsf{in}\, e' \quad \overset{\text{def}}{=} \quad (\lambda\, (x, p : x = e).\, \mathsf{subst}_{\,p}\, e')\; e\; (\mathsf{refl}\; e)$$

Here, $x = e$ *is the* identity type*, a type representing the fact that two expressions are equivalent. The elimination form* $\mathsf{subst}_{\,p}\, e'$ *essentially substitutes* x *by* e *in the type of* e' *when* $p : x = e$*. The introduction form* $\mathsf{refl}\; e$ *can only introduce a proof that* $e = e$*, i.e.,* $\mathsf{refl}\; e : e = e$

However, defining let this way introduces significant indirection in what we're trying to express, and creates unnecessary function applications which could complicate compilation, so I just add definitions to ECC^D.

In Figure 2.4, I define *definitional equivalence* (also known as *judgmental equivalence*) $\Gamma \vdash e \equiv e'$, which defines when two expressions are equivalent. Equivalence is defined as conversion up to η-equivalence on functions, which is standard (Luo, 1990).

Note that η-equivalence rules do not look like the standard η-equivalence for functions presented in Section 2.1, and do not actually check both sides are functions. For example, in Rule $\equiv$-η_1, the right side e_2 is just converted to an arbitrary e'_1 and then

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

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

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

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

Figure 2.4: ECC^D Equivalence

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

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

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

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

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

Figure 2.5: ECC^D Subtyping

applies e_1' as a function. This trick allows us to define η-equivalence without making reduction, conversion, or equivalence type directed, but still ensure that two expressions are η-equivalent only when both behave as equivalent functions. Note, however, that it ignores domain annotations, so the following apparently strange equivalence holds for arbitrary A and B.

$$\lambda\, x : A.\, e \equiv \lambda\, x : B.\, e$$

This is not a problem, since the type system will only use the equivalence relation *after* checking that the terms are well-typed. In fact, this design choice is an advantage, as discussed in Chapter 3. Because there is no explicit symmetry rule, we require two symmetric η-equivalence rules.

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

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{bool} : \mathsf{Prop}}\ \textsc{Bool} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \textsc{True} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False}$$

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

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

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

$\boxed{\vdash \Gamma}$

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

Figure 2.6: ECC^D Typing (excerpts)

I include cumulativity through *subtyping* $\Gamma \vdash A \preceq B$, Figure 2.5. Subtyping extends equivalence with the cumulativity rules, allowing types in lower universes to inhabit higher universes, and the standard congruence rules. Note that functions are not contravariant in their argument, but equi-variant. This is surprising and not strictly necessary; it is used to support a simpler set-theoretic semantics.[4] However, it is standard (Luo, 1990), so I stick with it.

An excerpt of type system for ECC^D in is given in Figure 2.6, and is mutually defined with the well-formedness relation on typing environments. The typing rules are essentially the same as presented earlier in Section 2.1, with the addition of the typing rule for let, and the rule for subtyping, Rule Conv. The complete figure is given in Figure A.6 in Appendix A. Rule Let gives a dependent type for let expressions; note that in $\mathsf{let}\ x = e\ \mathsf{in}\ e'$, the type $B[e/x]$ contains the sub-expression e. But it also binds $x = e$ as a definition while type checking the body e'. This essentially allows the dependency $B[e/x]$ to be resolved "early", as discussed earlier. Rule Conv allows any expression e whose type is A to also be valid at the super type B, or, since subtyping extends equivalence, when $A \equiv B$. Because equivalence and subtyping are defined on

4 https://web.archive.org/web/20180913043136/https://pigworker.wordpress.com/2015/01/09/universe-hierarchies/

$$\begin{array}{lrcl} \textit{Source Observations} & \mathsf{v} & ::= & \mathsf{true} \mid \mathsf{false} \\ \textit{Target Observations} & \mathbf{v} & ::= & \mathbf{true} \mid \mathbf{false} \end{array}$$

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

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

Figure 2.7: Source and Target Observations

untyped syntax, we must also ensure that B is a valid type. The well-formedness rules for environments are entirely standard.

2.2.2 Evaluation and Compilation

Typographical Note. *In this section, I use a* **bold, red, serif font** *for target language observations. For now, the target language is undefined, but the observations will be the same in every target language.*

An expression being type checked is different from *program*, *i.e.*, an expression that can be evaluated at run time. Expressions have arbitrary free variables and arbitrary types. *Programs* are closed expressions that evaluate and produce easily distinguished outputs. Programs must be closed or evaluation cannot proceed past some variables, and must have outputs that a programmer can distinguish. For example, a programmer does not want to try to distinguish between two functions, but can easily distinguish between two booleans or two printed strings.

To formalize programs we first need to fix a notion of *observation*, the distinguishable results of evaluating a program. Since ECC^D is effect-free and strongly normalizing, I define observations as boolean values in Figure 2.7. Observations should be simple for the programmer to distinguish even across languages. I formalize this by defining a cross-language observation relation $\approx$. This relates source and target observations in standard way—booleans are related across languages when they are the same. The languages will change for each translation, but all language observations will also be boolean values and use the same observation relation.

$\boxed{\Gamma \vdash \mathsf{e}}$ $\qquad\qquad$ $\boxed{\vdash \mathsf{e}}$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool}}{\Gamma \vdash \mathsf{e}} \qquad\qquad \frac{\cdot \vdash \mathsf{e}}{\vdash \mathsf{e}}$$

Figure 2.8: ECC^D Components and Programs

I formalize programs and program components via the judgments $\vdash \mathsf{e}$ and $\Gamma \vdash \mathsf{e}$ in Figure 2.8. A well-formed *program component* (or just *component*) is a well-typed expression e that produces an observation when linked with a closing substitution (*i.e.*, has type bool). A well-formed program is simply a closed component.

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

$$\mathsf{eval}(\mathsf{e}) \quad = \quad \mathsf{v} \quad \text{if} \vdash \mathsf{e} \text{ and } \cdot \vdash \mathsf{e} \rhd^* \mathsf{v}$$

Figure 2.9: ECC^D Evaluation

In Figure 2.9, I define the *evaluation (meta-)function* $\mathsf{eval}(\mathsf{e})$, which runs programs to produce observations. This defines the run-time semantics of ECC^D programs. It is this semantics that the compiler will preserve to prove compiler correctness, as discussed in Chapter 3.

$$\textit{Closing Substitutions} \quad \gamma \quad \overset{\text{def}}{=} \quad \cdot \mid \gamma[\mathsf{x} \mapsto \mathsf{e}]$$

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

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

$\boxed{\gamma(\mathsf{e}) = \mathsf{e}}$

$$\cdot(\mathsf{e}) = \mathsf{e} \qquad\qquad \gamma[\mathsf{x} \mapsto \mathsf{e}'](\mathsf{e}) = \gamma(\mathsf{e}[\mathsf{x}/\mathsf{e}'])$$

Figure 2.10: ECC^D Closing Substitutions and Linking

For reasoning about separate compilation, I formalize linking for ECC^D in Figure 2.10. Free variables Γ represent the imports to a component e, and their types represent the interface for those imports. A closing substitution γ represents the components that e is linked with via $\gamma(\mathsf{e})$. This is a meta-application of the closing substitution to e and replaces each $\mathsf{x} \in \Gamma$ by an expression $\gamma\mathsf{x}$ in the closing substitution. For linking to be well-defined—*i.e.*, for there to be no linking errors—the closing γ substitution must be well-typed, $\Gamma \vdash \gamma$, with respect to the same environment as e, $\Gamma \vdash \mathsf{e} : \mathsf{A}$. Typing for a substitution γ simply checks that each component $[\mathsf{x} \mapsto \mathsf{e}'] \in \gamma$ in the substitution has the corresponding type in Γ, *i.e.*, $\mathsf{e}' : \mathsf{A}$ when $\mathsf{x} : \mathsf{A} \in \Gamma$. For definitions in Γ, the substitution must map to a closed version of the same definition.

3 TYPE-PRESERVING COMPILATION

In this chapter, I introduce type-preserving compilation at a high level, the particular problems that arise in proving type preservation for dependent types, and the standard proof architecture I will use in the rest of this dissertation.

Typographical Note. *In this chapter I typeset a source language in a* blue, non-bold, sans-serif font*, and a target language in a* **bold, red, serif font***. Unless otherwise specified, these languages are arbitrary except for the requirements presented in Chapter 2, since the proof architecture and lessons apply generally to dependently typed source and target languages.*

3.1 A Brief History

I begin with a brief history of type preservation leading up to this dissertation. For a more in-depth survey of the use this early history of types in compilation, see Leroy (1998, Section 2 and 3).

Early compilers preserved types into intermediate languages (*ILs*) for optimization, not for ruling out errors. Aho et al. (1986, Chapter 10) describe simple optimizations that use types to specialize arithmetic operations, such as generating different instructions for integer or floating-point numbers based on types. Leroy (1992) uses types in an intermediate language to remove memory indirection introduced to support polymorphic functions. Tarditi et al. (1996) develop *TIL*, a compiler for ML with type-directed optimizations as a central feature. The TIL compiler preserved types down to a typed intermediate language (hence the name TIL), in which types are used to optimize polymorphic functions, loops, and garbage collection. The Glasgow Haskell Compiler (*GHC*) employs a typed intermediate language, *Core*, and performs some type-based optimizations in Core (Peyton Jones, 1996). The *SML/NJ* compiler was extended with a typed intermediate language, *FLINT*, to support type-based optimization and study how to reduce compilation overhead caused by preserving types (Shao et al., 1998). The System F to Typed Assembly Languages (*F-to-TAL*) compiler was preserved all typing structure down to a typed assembly language (*TAL*), and leverages types in the assembly language for low-level optimizations such as array bounds check elimination (Morrisett et al., 1999).

While initially used for optimization, early work on type-preserving compilation of functional languages recognized the use of types for ruling out some miscompilation errors. For example, the TIL compiler, while focusing on the use of types for optimization, points out that the ability to type check the output of the compiler helps to find and fix miscompilation errors (Tarditi et al., 1996). The GHC team also recognized the potential for ruling out miscompilation errors and developed a tool, Core Lint, for type checking the output of each compiler pass. This use seemed to surpass other uses; as the author, Simon Peyton-Jones, says "One of the dumbest things we did was to delay writing Core Lint." (Peyton Jones, 1996, section 8). The F-to-TAL compiler uses types in the target assembly language and proves type safety, ruling out a class of miscompilation errors from the entire compiler in the process: type and memory safety errors are statically ruled out (Morrisett et al., 1999).

This work spawned an approach related to type preservation for ruling out both linking errors and miscompilation errors, in particular, *proof-carrying code*. The work on TIL is credited (Tarditi et al., 2004) with spawning the idea of proof-carrying code (PCC) (Necula, 1997) and certifying compilation (Necula and Lee, 1998). The two ideas are essentially similar. In PCC, the idea is to pair implementation code (executable) with a correctness specification and a proof that the implementation satisfies its specification. This allows ruling out linking errors by enforcing the specification when linking, and removes the implementation from the trusted code base. Certifying compilation extends this idea to compilers to remove the compiler from the trusted code base and eliminate miscompilation errors. In a *certifying compiler*, rather than prove the compiler correct, we design the compiler to produce both proof-carrying code, *i.e.*, to produce or preserve *certificates* (proofs) that the compiled code meets its correctness specifications.

PCC and certifying compilation can be viewed as instances of type preservation in which we interpret type broadly as either intrinsic or extrinsic, *i.e.*, as either Church style or Curry style. Work on PCC and certifying compilation proceeded two veins: (1) using intrinsically typed languages and (2) using extrinsically typed languages.

In *intrinsically typed systems*, such as in TIL and TAL, where types represent specifications, well-typed implementations represent proofs and type checking is proof checking. The work on TIL describes an untrusted compiler that could produce fully optimized TIL code and a client could automatically verify simple safety properties, such as memory safety, by type checking (Tarditi et al., 1996). This guarantees absence of memory safety errors introduced when linking two compiled components, *i.e.*, it rules out linking errors that violate memory safety. In TAL, the idea is carried out to the assembly level. Any two TAL programs are guaranteed to be free of memory safety errors when linked, if their types are compatible. Later work introduced *DTAL* (Xi and Harper, 2001), a variant of TAL with indexed types, which could statically rule out memory indexing errors. Chen et al. (2010) develop a certifying compiler from *Fine*, an ML-like language with refinement types, to *DCIL*, a variant of the .NET Common Intermediate Language with refinement types, capable of certifying functional correctness properties, such as access control and information-flow policies.

In *extrinsic type systems*, such as the work of Necula (1997) and Necula and Lee (1998), specifications and proofs are written in separate languages from the implementation. Necula (1997) uses LF to represent specifications and proofs, and axiomatizes the behavior of implementation primitives in LF specifications. Shao et al. (2005) take a similar approach, but use CIC as an extrinsic type system over a high-level functional language.

Compared to extrinsic systems, intrinsic systems have smaller certificates by reusing the implementation to represent the proof (Xi and Harper, 2001), and can avoid the problem of specification puns. However, it is simpler to build certificates in extrinsic systems, since the specification and proof can be designed and manipulated separately from the implementation.

So far, work on certifying compilation and PCC has been limited to either extrinsic systems or to restricted (*i.e.*, non-full-spectrum) dependent types. This is apparently due to the difficulty of automatically transforming proofs during compilation. For example, Shao et al. (2005) point to using an extrinsic system explicitly because of a known impossibility result regarding preserving dependent types through the standard CPS translation (Barthe and Uustalu, 2002).

3.2 A Model Type-Preserving Compiler

F-to-TAL developed the standard model of a type-preserving compiler from a high-level functional to a low-level assembly language. This work uses System F as a stand-in for a functional source programming language. The target language, TAL, is a low-level assembly-like language. F-to-TAL is structured as five passes, depicted in Figure 3.1. The first two are so-called “front-end” translations, which make high-level functional control-flow and data-flow explicit in order to facilitate low-level transformations. These are the continuation-passing style *(CPS)* translation, and the closure conversion translation. I’ll explain these two in detail shortly. The next three so-called “back-end” translations make machine-level details explicit. These are: hoisting, a simple administrative pass that lifts function definitions to the top-level; explicit allocation, which makes the heap and memory operations explicit; and code generation, which makes machine details like word size and registers explicit, and which implements high-level operations in terms of sets of machine instructions.

In this dissertation, I focus on the two front-end translations as they introduce a particular challenge for type-preserving compilation. They are standard translations for studying type preservation (Minamide et al., 1996; Barthe et al., 1999; Shao et al., 2005; Ahmed and Blume, 2008; Chen et al., 2010; Ahmed and Blume, 2011; New et al., 2016). As I describe in the rest of this section, the translations fundamentally change the meaning of types, particularly for higher-order expressions, by introducing explicit distinctions into the terms described by types. By creating new distinctions during compilation, a type-preserving compiler needs to automatically adapt specifications and proofs to describe and proof the correctness of code with explicit distinctions.

Often we explain CPS in terms of making control flow and evaluation order explicit, but I think of it as changing each type by imposing an explicit distinction between *computations*, which evaluate at run time and may have effects, and *values*, which simply exist and do not evaluate at run-time until they are composed with a computation. In

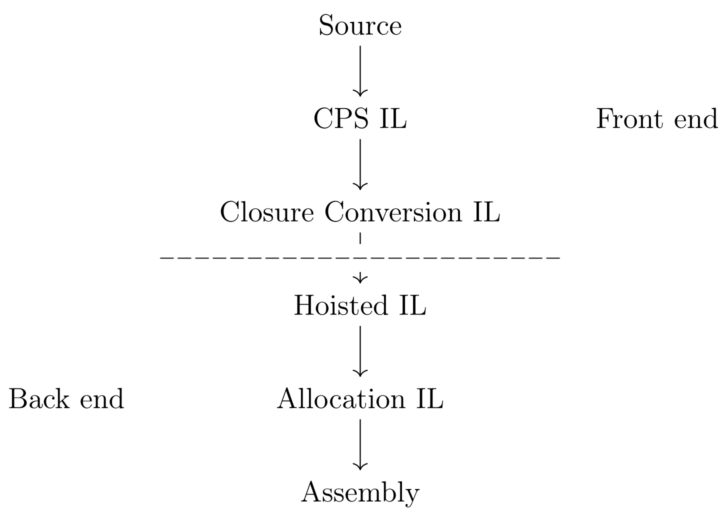

Figure 3.1: Model Type-Preserving Compiler

most functional languages, prior to CPS translation, all terms are implicitly cast to values through evaluation; this is helpful for equational reasoning. CPS translation transforms every term into a computation that expects a continuation to which it passes the underlying value when the computation is complete. This facilitates low-level machine implementation since machines typically have such distinction, for example, with values that exist in the heap and in registers and instructions (computations) that manipulate those values.

To understand this, consider the following example. We might write the expression $e_1 = (\lambda x. x)$ true in ECC^D which we think of as being equivalent to the value $e_2 =$ true. This is *equivalent* to true, but e_1 *computes* while e_2 does not, so we need to make that distinction explicit. Eventually, we need to implement e_1 as something like the following pseudo-assembly, whereas we implement e_2 as `true`.

```
// expects value in x and continuation in k
f:
  ret = x;
  goto k;
halt:
  print(ret);
main:
  k = halt;
  x = true;
  goto f;
```

When the program begins, it will initialize the continuation to `halt`, which will end the program with the final observation. Then it will set a register or stack frame, `x`, to the value `true`, and jump to the function `f`. Notice that $\lambda x. x$ makes no explicit distinctions between value and computation, but to generate the pseudo-assembly we need to know that the body of the function `x` is not merely a value but a computation that will return somewhere, unlike `true` which is merely a value to be passed around.

There are alternatives to CPS translation that make the same value vs computation distinction, and each offer advantages and disadvantages for type preservation and compilation. For example, the A-normal form (*ANF*) and the monadic-normal form translations also impose this distinction. Unlike CPS, ANF and monadic form do not make explicit changes to types but primarily modify syntactic structure. I discuss this further in Chapter 4, but in short: by avoiding changing types significantly we can simplify type preservation.

The *closure-conversion* translation is typically explained as about making first-class functions easy to heap allocate, but, as in CPS, I prefer to view it as introducing an explicit distinction—this time, a distinction between a *use* of computation and a *definition* of a computation. This is usually seen in functions because functions are the only source-level abstraction for defining computations in many languages. Prior to closure conversion, every first-class function can simultaneously define a new computation and can be used immediately.

For example, in $\mathsf{e}_1 = (\lambda\,\mathsf{x}.\,\mathsf{x})\ \mathsf{true}$ from before, the function $\lambda\,\mathsf{x}.\,\mathsf{x}$ is simultaneously *defined* (*i.e.*, it comes into existence), and *used* (*i.e.*, it is immediately applied). After closure conversion, all computation is represented by closed *code*, which defines the computation generally and separately from its use. In the pseudocode example above, we need to somehow separate the definition of $(\lambda\,\mathsf{x}.\,\mathsf{x})$ as a labeled block of code `f:`, and its use, *i.e.*, the **`goto`** `f;`.

The primary difficultly in closure conversion is that defining a computation (*e.g.*, a function) *implicitly* captures the local environment, and using the computation *implicitly* gives access to that local environment, regardless of where the computation is used. To make the definition and use explicitly distinct, we must make this *implicit* environment *explicit*. We do this by introducing an *explicit closure* object, *i.e.*, closed code paired with the local environment the computation expects. The code part can be lifted to the top-level while each use explicitly passes in the environment.

The main alternative to closure conversion is defunctionalization. While closure conversion represents the implicit environment as an explicit object that is passed to closed code, defunctionalization transforms every application into a case-split, and every closure into a constructor for a sum. The constructor holds the local environment and the case-split is responsible for setting up the local environment in the scope of the code. This ensures that the local environment cannot be passed to untrusted code, which would leak hidden local variables. Unfortunately, it requires a whole-program transformation, so it cannot be used with separate compilation.

The goal of this dissertation is to build a model of the front-end translations of a compiler. This solves the main challenges of transforming higher-order dependently typed expressions, and informs the design of dependent-type-preserving translations in general. After these two translations, we end up in a language analogous to a pure subset of C with all higher-order expressions essentially encoded in first-order data.

Then the challenges become about representing dependently typed machine concepts, such as the heap and word sized registers.

3.3 The Difficulty of Preserving Dependency

Generally speaking, type preservation is difficult because when introducing new distinctions we must automatically adapt specifications and proofs. After closure conversion, specifications and proofs about a function with certain lexical variables must be adapted to be valid in the global scope. After CPS, a value of type A has a fundamentally different interface from computations of type A. Any specification that relied on expressions being implicitly cast to values will need some explicit way to transform a computation into a value.

The problem is worse for intrinsic dependent types because the compiler transformations disrupt the *syntactic* reasoning used by the type system to decide type checking. To allow arbitrary low-level run-time terms to still appear in types, the type system must be able to check proofs encoded in low-level abstractions. For example, in ECC^D, the type system decides equivalence by evaluating run-time terms during type checking. This works well in high-level, functional languages such as the core language of Coq, but evaluating low-level machine languages requires threading all the machine-state (*e.g.*, the heap or register file) through evaluation, so we must design a type system that captures all of that necessary state. We would need to adapt all the typing judgments—reduction, conversion, equivalence, subtyping, typing, and well-formedness—to track this state and capture new invariants that are expressed explicitly at the low-level.

Each of the features of dependency introduced in Chapter 2 rely on this kind of syntactic reasoning, which is disrupted by translation. For example, Barthe and Uustalu (2002) show that the standard call-by-name (*CBN*) double-negation CPS translation is not type preserving in the presence of dependent pairs. The problem is that the typing rule Rule Snd for second projection of a dependent pair, $\mathsf{snd}\,\mathsf{e} : \mathsf{B}[\mathsf{fst}\,\mathsf{e}/\mathsf{x}]$, copies the syntax $\mathsf{fst}\,\mathsf{e}$ into the type B. This represents evaluating e to a value and taking the first projection. After CPS translation, such a term is invalid as there is a computation vs value distinction, so target language version of the typing rule Rule Snd can no longer express dependency on a computation *syntactically*, the way it was expressed in the source language. We end up needing some syntactic way to express $\mathsf{snd}\,\mathsf{e} : \mathsf{B}[\text{as-a-value}(\mathsf{fst}\,\mathsf{e})/\mathsf{x}]$, that is, to cast an arbitrary computation to a value but in a way that is compatible with the low-level machine semantics we want the output to use. (I discuss this example in greater detail in Chapter 6.)

There is an additional difficulty in *proving* type preservation in the presence of dependent types—the proof architecture. I discuss this difficulty in detail next.

3.4 Proving Type Preservation for Dependent Types

In this section, I introduce the proof architecture I use in the rest of this paper, discuss some difficulties that arise when attempting to prove type preservation for dependent types, and how this particular architecture and the design of ECC^D avoids them.

Typographical Note. *Recall that the source language in a* blue, non-bold, sans-serif font*, and target language in a* **bold, red, serif font***, represent arbitrary source and target languages.*

Ultimately, our goal is to prove type preservation of some translation $[\![\mathsf{e}]\!]$. Semi-formally (since so far this translation is undefined), this is stated in Theorem 3.4.1.

Theorem 3.4.1 (Type Preservation). *If* $\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A}$ *then* $[\![\mathsf{\Gamma}]\!] \vdash [\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]$.

Type preservation states that if, in the source type system, the source term e is well-typed with type A under the local environment $\mathsf{\Gamma}$, then, in the target type system, the translation $[\![\mathsf{e}]\!]$ of e is well-typed at the translated type $[\![\mathsf{A}]\!]$ under the translated local environment $[\![\mathsf{\Gamma}]\!]$.

The proof of type preservation is typically by induction on the same *structure* over which the compiler is defined. What this structure is depends on what information the compiler requires to perform a transformation.

In an untyped setting, that *structure* is the structure (or size) of syntax. For example, the untyped call-by-value CPS translation of Plotkin (1975) on the untyped λ-calculus is given below.

$$
\begin{aligned}
[\![x]\!] &\overset{\text{def}}{=} \lambda\, k.\, k\ x \\
[\![\lambda\, x.\, e]\!] &\overset{\text{def}}{=} \lambda\, k.\, k\ (\lambda\, x.\, [\![e]\!]) \\
[\![e_1\ e_2]\!] &\overset{\text{def}}{=} \lambda\, k.\, [\![e_1]\!]\ (\lambda\, x_1.\, [\![e_2]\!]\ (\lambda\, x_2.\, x_1\ x_2\ k))
\end{aligned}
$$

Note that the translation is easily defined by induction on the syntactic structure of terms.

In a typed setting or in a type-preserving setting, the compiler must often be defined over the typing derivation or the height of typing derivations. This is because, in general, the compiler may need to make decisions based on types, or produce type annotation in the target language. For example, if we try to adapt the above call-by-value CPS translation to simply-typed λ-calculus functions, we wind up with the following incomplete translation.

$$
[\![\lambda\, x : A.\, e]\!] \overset{\text{not-really-def}}{=} \lambda\, k : B'.\, k\ (\lambda\, x : [\![A]\!].\, [\![e]\!])
$$

The target language, being typed, requires a type annotation B' for the continuation k. To produce this annotation, the compiler requires the type $A \to B$ of the function $\lambda\, x : A.\, e$. Instead, the translation should be defined over typing derivations, as follows.

$$\frac{\Gamma, x : A \vdash e : B \rightsquigarrow e^+}{\Gamma \vdash \lambda\, x : A.\, e : A \to B \rightsquigarrow \lambda\, k : (\llbracket A \to B \rrbracket \to \bot).\, k\ (\lambda\, x : \llbracket A \rrbracket.\, e^+)}$$

This way, the translation has access to all typing information in order to produce type annotations or use type information to inform the translation.

In a simply-typed setting, defining compilation over typing derivations poses no problems. We can define the translation $\llbracket \mathsf{e} \rrbracket$ of terms as short-hand, defined as follows.

$$\llbracket \mathsf{e} \rrbracket \overset{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow \mathbf{e}$$

The translation over syntax $\llbracket \mathsf{e} \rrbracket$ is simply notation for the translation over typing derivations, and we make the typing derivation $\Gamma \vdash \mathsf{e} : \mathsf{A}$ an implicit parameter. Then we prove Theorem 3.4.1 by induction over the typing derivation.

With dependent types, this simple recipe does not work because we must show many mutually defined judgments are preserved, not just one typing judgment. The problem is due to "the infernal way that everything depends on everything else"[1]. The judgmental structure of the type system is much more complex—recall that we defined six judgments just to define the ECC^D type system—and the judgments can be mutually defined. To prove type preservation, we must prove that each additional judgment is preserved. Because those judgments are mutually defined, we cannot, in general, cleanly break up the judgmental structure into separate lemmas—we may have to show that all judgments are preserved in one large simultaneous induction. I'll introduce each of the lemmas required by the judgmental structure of ECC^D, before going into detail about the problems of proving each judgment is preserved.

3.4.1 The Key Lemmas for Type Preservation

As the judgmental structure of typing gets more complex, we need additional lemmas before we can prove Theorem 3.4.1.

For example if the type system allows substitution into types, we need a *compositionality* lemma, which states that translating first and then substituting is equivalent to substituting first and then translating.

Lemma 3.4.2 (Compositionality). $\llbracket \mathsf{A}[\mathsf{A}'/\mathsf{x}] \rrbracket \equiv \llbracket \mathsf{A} \rrbracket[\llbracket \mathsf{A}' \rrbracket/\mathbf{x}]$

This requires some definition of type equivalence, $\equiv$, which could be as simple as syntactic identity. For dependent types, it must usually be definitional equivalence. Intuitively, this is because a dependent-type-preserving translation must introduce new

1 From McBride (2010).

equivalence rules for whatever feature is made explicit by the translation. (I discuss this further in Chapter 8.)

In ECC^D, we require compositionality, due to Rule App, Rule Snd, and Rule Let. For example, consider Rule App.

$$\frac{\Gamma \vdash e : \Pi x : A'. B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e\ e' : B[e'/x]}\ \textsc{App}$$

We know that $\Gamma \vdash e_1\ e_2 : B[e_2/x]$, and must show that $[\![\Gamma]\!] \vdash [\![e_1\ e_2]\!] : [\![B[e_2/x]]\!]$. Even in the case that the translation is simply homomorphic on application, thus leaves application alone, the target variant of Rule App will only tell us $[\![\Gamma]\!] \vdash [\![e_1]\!]\ [\![e_2]\!] : [\![B]\!][[\![e_2]\!]/\mathbf{x}]$. Thus, we either need to know $[\![B[e_2/x]]\!]$ and $[\![B]\!][[\![e_2]\!]/\mathbf{x}]$ are syntactically identical or, by Rule Conv, equivalent. In general, they will not be syntactically identical. For example, consider closure conversion, which is sensitive to the free variables. If translating the function $e = \lambda y. x$, we end up with the following two translations depending on whether we translate before or after substitution.

$$[\![e]\!][[\![e_1]\!]/x] = (\langle \lambda\, y, n.\ \mathsf{fst}\ n, \langle x \rangle \rangle)[[\![e_1]\!]/x] = (\langle \lambda\, y, n.\ \mathsf{fst}\ n, \langle [\![e_1]\!] \rangle \rangle)$$

$$[\![e[e_1/x]]\!] = \langle \lambda\, y, n.\ [\![e_1]\!], \langle\rangle \rangle$$

Translating before substitution will yield a closure environment in which the expression $[\![e_1]\!]$ is substituted in for x, while translating after substitution will yield an environment with all free variables. (I discuss this example further in Chapter 5.)

If the type system has a type equivalence judgment then we must first prove that *equivalence preservation*, *i.e.*, that type equivalence is preserved, and then prove type preservation,

Lemma 3.4.3 (Preservation of Equivalence). *If* $\Gamma \vdash A \equiv B$ *then* $\Gamma \vdash [\![A]\!] \equiv [\![B]\!]$

This states that if A and B are equivalent in the source, then $[\![A]\!]$ and $[\![B]\!]$ are equivalent in the target.

In the Calculus of Constructions (CoC), similar to ECC^D but without subtyping, we require equivalence preservation due to Rule Conv.

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : B}\ \textsc{Conv}$$

When proving type preservation by induction on the typing derivation, we must consider the case of Rule Conv: intuitively, for some translation, we will know by the induction hypothesis that $[\![e]\!]$ has type $[\![A]\!]$, and must show that $[\![e]\!]$ has type $[\![B]\!]$. This follows easily by Rule Conv in the target language if $[\![B]\!] \equiv [\![A]\!]$, which we know if we have equivalence preservation.

Depending on how equivalence is defined, we may need further supporting lemmas. For example, in ECC^D, we implement equivalence as conversion up to η-equivalence.

Therefore, to show equivalence is preserved, we need to show reduction and conversion are preserved (up to equivalence, since in general we don't care that expressions reduce or convert to *syntactically* the same, only that they reduce to some equivalent expression).

Lemma 3.4.4 (Preservation of Reduction). *If* $\Gamma \vdash e \triangleright e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \triangleright^* \mathbf{e'}$ *and* $[\![\Gamma]\!] \vdash \mathbf{e'} \equiv [\![e']\!]$

This lemma is similar to compiler correctness theorems in that we don't want to require the translated term $[\![e]\!]$ to simulate the reductions of e in lock-step. Instead, we specify this as a weak-simulation: it is sufficient for $[\![e]\!]$ to be convertible to some $\mathbf{e'}$ that is equivalent to $[\![e']\!]$.

Lemma 3.4.5 (Preservation of Conversion). *If* $\Gamma \vdash e \triangleright^* e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \triangleright^* \mathbf{e'}$ *and* $[\![\Gamma]\!] \vdash \mathbf{e'} \equiv [\![e']\!]$

In a simply typed language, we could prove this lemma by induction on the length of reduction sequences. However, the judgmental structure of conversion is not always so simple. Recall that in ECC^D, the congruence rule for let expressions, Rule Red-Cong-Let, introduces a definition during conversion, *i.e.*, we have the following rule.

$$\frac{\Gamma \vdash e_1 \triangleright^* e_1' \qquad \Gamma, x = e' \vdash e_2 \triangleright^* e_2'}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \triangleright^* \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2'}\ \textsc{Red-Cong-Let}$$

This means that proving conversion is preserved must be by induction on the conversion derivation, so we correctly capture the definition.

By specifying equivalence via reduction, we get an extra benefit: the three previous lemmas, required for type preservation, also easily imply compiler correctness, as I discuss shortly in Section 3.4.3.

If the type system has a subtyping judgment, then we must also prove that subtyping is preserved.

Lemma 3.4.6 (Preservation of Subtyping). *If* $\Gamma \vdash A \preceq B$ *then* $\Gamma \vdash [\![A]\!] \preceq [\![B]\!]$

This states that if A is a subtype of B in the source, then $[\![A]\!]$ is still a subtype of $[\![B]\!]$ in the target.

In ECC^D, we extend conversion to subtyping to support cumulativity. We have the following Rule Conv.

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

Similar to the discussion for Lemma 3.4.3, the proof of type preservation will require showing that $[\![e]\!]$ has type $[\![B]\!]$, given that $[\![e]\!]$ has type $[\![A]\!]$ and $A \preceq B$. By the target language Rule Conv and preservation of subtyping, the proof is simple. However, for certain translations, proving that subtyping is preserved can be challenging. For example, the standard locally polymorphic answer type CPS translation relies on impredicativity, so it translates any computationally relevant type $\mathsf{A} : \mathsf{Type}_i$ into the type $\mathbf{\Pi}\, \boldsymbol{\alpha} : \mathbf{Prop}\,.\, (\mathsf{A}^{+} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha} : \mathbf{Prop}$. I discuss this translation further in Chapter 6, but the important thing to observe now is that A exists in a higher universe, while $[\![\mathsf{A}]\!]$ *must* exist in the base universe $\mathbf{Prop}$. This does not preserve subtyping.

With all of these lemmas, we will finally be able to prove that the typing and well-formedness relations are preserved. The theorem we ultimately want is Theorem 3.4.1, we must prove it via something like the following lemma.

Lemma 3.4.7 (Type and Well-formedness Preservation)**.**

1. *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]$
2. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]$

Since well-formedness of environments and typing are mutually defined, we have to prove these two by simultaneous induction on the mutually defined judgments.

3.4.2 The Problem with Typed Equivalence

The proof architecture presented in the previous section does not work if equivalence is mutually defined with typing. This can happen with some formulations of dependent types with strong η-principles. For example, we might want the following η-equivalence for the unit type.

$$\frac{}{\Gamma \vdash \mathsf{e} \equiv \langle\rangle : 1}$$

That is, every expression e should be equivalent to the unit value $\langle\rangle$ at the unit type 1. This requires that equivalence be defined over well-typed expressions.

Unfortunately, defining equivalence over well-typed expressions introduces a circularity into the proof architecture for type preservation. If equivalence is typed, then we *must* show type preservation *before* we can establish any equivalence about translated expressions. But establishing type preservation requires showing equivalence is preserved.

For example, as discussed earlier, we require Lemma 3.4.2 (Compositionality) to prove Theorem 3.4.1 (Type Preservation). But, compositionality must, in general, be proven in terms of equivalence. If equivalence is typed, then we must show *type preservation before compositionality*. But, since some type rules are defined by substitution, we must show *compositionality before type preservation*. That is, we must:

- prove type preservation before compositionality, and
- prove compositionality before type preservation.

Similarly, as discussed before, we require Lemma 3.4.3 (Preservation of Equivalence) in order to show type preservation. But if equivalence is typed, we cannot show preservation of equivalence until we show type preservation. So we must:

- prove type preservation before equivalence preservation, and
- prove equivalence preservation before type preservation.

And if equivalence is defined using substitution, we cannot show preservation of equivalence until we show compositionality. But compositionality requires establishing equivalence, which requires type preservation. We must:

- prove compositionality before equivalence preservation, and
- prove equivalence preservation before type preservation, and
- prove type preservation before compositionality.

To see the problem concretely, consider the CPS translation by double negation in the presence of type equivalence rule Rule Conv, but with a typed definition of equivalence. Because CPS translation adds type annotations for continuations, we can end up with the following translation.

$$\frac{\Gamma \vdash \mathsf{B} : \mathsf{U} \rightsquigarrow \mathbf{B} \qquad \Gamma \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow \mathbf{e}_B \qquad \Gamma \vdash \mathsf{A} : \mathsf{U} \rightsquigarrow \mathbf{A}}{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow \boldsymbol{\lambda}\,\mathbf{k} : (\mathbf{A} \to \bot).\,\mathbf{e}_B\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{k}\ \mathbf{x})}$$

That is, when translating Rule Conv, we produce a computation which takes a continuation $\mathbf{k}$ that expects some value of type $\mathbf{A}$, but then we call $\mathbf{k}$ with a value of type $\mathbf{B}$. Unless $\mathbf{A}$ and $\mathbf{B}$ are equivalent, the domain annotations on this term make little sense—why can we apply $\mathbf{k}$, which expects a $\mathbf{A}$, to $\mathbf{x}$ of type $\mathbf{B}$? So we must first show equivalence is preserved. To prove equivalence is preserved, we would need to show $\mathbf{e}_B$ is equivalent to $\mathbf{e}^+\boldsymbol{\lambda}\,\mathbf{k} : (\mathbf{A} \to \bot).\,\mathbf{e}_B\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,)\mathbf{k}\ \mathbf{x}$. If equivalence is typed, then we need to show that $\mathbf{e}^+$ and $\mathbf{e}_B$ are well-typed first.

Barthe et al. (1999) observe this problem in an early attempt at type-preserving CPS translation of dependent types. Their solution is to give up domain annotations, thus yielding undecidable type checking. This is unsatisfying if we want to rely on type checking for ruling out miscompilation and linking errors.

Instead, in this work, I require an untyped equivalence for the source and target languages. This is a strong requirement on the type theory, but not unrealistic; Coq relies on an untyped equivalence, for example. By using an untyped equivalence, we can prove equivalence between terms without regard to their domain annotations, and the above translation of Rule Conv can be easily shown equivalent to $\mathbf{e}_B$ (by η-equivalence). This breaks the circularity, and allows us to stage all the lemmas from the previous section.

1. Lemma 3.4.2 (Compositionality)
2. Lemma 3.4.4 (Preservation of Reduction)
3. Lemma 3.4.5 (Preservation of Conversion)

4. Lemma 3.4.3 (Preservation of Equivalence)
5. Lemma 3.4.6 (Preservation of Subtyping)
6. Lemma 3.4.7 (Type and Well-formedness Preservation)
7. Theorem 3.4.1 (Type Preservation)

A possible alternative is to prove all the lemmas simultaneously by one large mutual induction, as stated below. I have not investigated this approach, as the sheer number of cases and induction hypotheses to keep straight in one theorem seems daunting. [2]

Conjecture 3.4.8 (Type Preservation (with Typed Equivalence)).

1. $[\![\Gamma]\!] \vdash [\![e[e'/x]]\!] \equiv [\![e]\!][[\![e']\!]/x]$
2. *If* $\Gamma \vdash e \equiv e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$
3. *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]$
4. *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$

3.4.3 Type Preservation and Compiler Correctness

On its own, type preservation is a weak theorem, but the proof architecture just presented and the structure of the dependent types essentially force us to prove a standard compiler correctness theorem: correctness of separate compilation. The statement of Theorem 3.4.1 (Type Preservation) only tells us that if the source term was well-typed, then so is the target term. A compiler *could* trivially satisfy this property by compiling every term to the unit value and every type to the unit type. It is unlikely to satisfy all the lemmas in Section 3.4, but it would still satisfy Theorem 3.4.1 (Type Preservation). To trust Theorem 3.4.1, we must *at least* add the translation of types to the trusted code base, *i.e.*, we must at least understand how types are translated. By reading the translation of types, we can have confidence that the translation is non-trivial. Even then, type preservation does not give many guarantees about how programs execute unless the types are complete, full-functional specifications. Thankfully, the judgmental structure of dependent types, and by defining equivalence in terms of conversion of open terms, type preservation forces us to prove a stronger compiler correctness theorem: that *linking* and *then evaluating* in the *source* is equivalent to translating and *linking* and *then evaluating* in the *target*.

To specify compiler correctness, we need an independent specification that tells us how observations are related to target observations, hence the definition of observations in ECC^D. For instance, when compiling to C we might specify that the number 5 is related to the bits $0x101$. Without a specification, independent of the compiler, there is

2 To investigate this approach, I suggest attempting this proof technique, with the help of a proof assistant, for the abstract closure conversion in Chapter 5.

no definition that the compiler can be correct with respect to. The best we could prove is that the *translation* of the value v produced in the source is *definitionally equivalent* to the value we get by running the translated term, *i.e.*, we would get $[\![\mathsf{v}]\!] \equiv \mathbf{eval}([\![\mathsf{e}]\!])$. This fails to tell us how $[\![\mathsf{v}]\!]$ is related to v, unless we inspect the compiler. Thankfully, it is simple to define related observations for dependently typed languages, since these languages are usually error-, divergence-, and effect-free.

We must also formalize linking for each language, as we did in Chapter 2 for ECC^D. We define well-typed closing substitution γ as in ECC^D, and extend the translation to closing substitutions point-wise.

Correctness of separate compilation, formally stated below, tells us that if e is a well-typed source component, and γ is a well-typed closing substitution, and linking γ with e (written $\gamma(\mathsf{e})$) then evaluating in the source produces an observation v that is related to the observation produced by separately compiling γ and e, linking, and then evaluating in the target.

Theorem 3.4.9 (Separate Compilation Correctness)**.** *If* $\Gamma \vdash \mathsf{e}$*,* $\Gamma \vdash \gamma$*, then* $\mathsf{eval}(\gamma(\mathsf{e})) \approx \mathbf{eval}([\![\gamma]\!]\,([\![\mathsf{e}]\!]))$.

The proof of this theorem will always follow essentially the same structure, and is expressed as the following commuting diagram.

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

This diagram in terms of equivalence suffices since $\equiv$ corresponds to $\approx$ on observations, *e.g.*, $\mathsf{true} \equiv \mathsf{true}$ iff $\mathsf{true} \approx [\![\mathsf{true}]\!]$ iff $\mathsf{true} \approx \mathbf{true}$. The diagram commutes since $\mathsf{eval}(\gamma(\mathsf{e})) \equiv \gamma(\mathsf{e})$, since evaluation and equivalence are both defined in terms of conversion, $\gamma(\mathsf{e}) \equiv [\![\gamma(\mathsf{e})]\!]$ by Lemma 3.4.3 (Preservation of Equivalence), $[\![\gamma]\!]\,([\![\mathsf{e}]\!]) \equiv [\![\gamma(\mathsf{e})]\!]$, by Lemma 3.4.2 (Compositionality), and $\mathbf{eval}([\![\gamma]\!]\,([\![\mathsf{e}]\!])) \equiv [\![\gamma]\!]\,([\![\mathsf{e}]\!])$, again because evaluation and equivalence are defined in terms of conversion.

This recipe will not be *exactly* the same in all languages. For example, in Chapter 4 I define a machine evaluation semantics separate from conversion in the target language. To derive compiler correctness from preservation conversion we must show the machine semantics coincide to or refine conversion.

This separate-compilation correctness theorem is similar to the guarantees provided by SepCompCert (Kang et al., 2016) in that it supports linking with only the output of the same compiler. We could support more flexible notions of linking—such as linking with code produced by different compilers, from different source languages, or code written directly in the target language—by defining an independent specification for when closing substitutions are related across languages (*e.g.*, (Neis et al., 2015; Ahmed and Blume, 2011; New et al., 2016; Perconti and Ahmed, 2014)). This is orthogonal to my thesis and I do not consider it further.

Of course, this correctness of separate compilation implies the usual "whole-program correctness" theorem, typically called just *compiler correctness*, stated below.

Corollary 3.4.10 (Whole-Program Correctness)**.** *If* $\vdash \mathsf{e}$*, then* $\mathsf{eval}(\mathsf{e}) \approx [\![\mathbf{eval}(\mathsf{e})]\!]$.

This simply states that any program evaluates to related observations before and after compilation.

3.5 Type Preservation as Syntactic Modeling

When designing new dependently typed (target) languages, we also need to prove type safety and consistency of the type system to have confidence that the proofs we have preserved are meaningful. There are many ways to do this: we can give a denotational or a categorical semantics, or prove progress, preservation and normalization of the operational semantics.

My preferred method is to reduce type safety and consistency to that of an existing type theory via a type-preserving translation, *i.e.*, by providing a *syntactic model* of the target theory in another theory. This use of type preservation is common in the literature. Bernardy et al. (2012) give a syntactic model of parametric Pure Type Systems (*PTS*) by translation into an existing PTS, and show how to develop a parametric model of CoC and use it to prove free theorems. Pédrot and Tabareau (2017) give a syntactic model of CIC with effects by translation into a CIC. Boulier et al. (2017) discuss this technique in detail and give several example syntactic models.

The ultimate theorems we wish to prove, type safety[3] and consistency, are stated below. Typically, we think of type safety when interpreting expressions as programs, and consistency when interpreting expressions as proofs.

Theorem 3.5.1 (Type Safety)**.** *If* $\vdash \mathbf{e}$ *then* $\mathbf{eval}(\mathbf{e})$ *is well-defined.*

Type safety states that any well-typed program "evaluates correctly". Recall from Chapter 2 that in this context, this means that evaluation produces a value.

Theorem 3.5.2 (Logical Consistency)**.** *There does not exist a* $\mathbf{e}$ *such that* $\cdot \vdash \mathbf{e} : \bot$*.*

Logical consistency tells us that there is no closed well-typed proof of $\bot$, the false theorem or uninhabited type.

To prove these two theorems, it suffices to give a type-preserving translation $[\![\mathbf{e}]\!]^{\circ}$ from target terms to the model language, and prove that falseness is preserved. To prove type preservation, we follow the same type-preservation recipe described earlier in Section 3.4. The statement of preservation of falseness is given below.

Lemma 3.5.3 (Preservation of Falseness)**.** $[\![\bot]\!]^{\circ} \equiv \bot$

This states that the definition of the false in the target is modeled as false in the model language.

[3] I use type safety as opposed to *type soundness*. *Soundness* has too many meaning in this context—*e.g.*, logical soundness—and *safety* helps us focus on the interpretation of this theorem as ensuring safe execution of programs.

The proof of each of the above theorems follows by a simple proof by contradiction. For example, consistency follows because, if the target were not consistent then we have a proof of $\perp$, and by type preservation and preservation of falseness, we could translate that proof into a proof of $\perp$ in the model, violating the consistency of the model. Since the model is consistent, the target language must be as well.

4 A-NORMAL FORM

In this chapter, I develop the first of the two front-end translations for a type-preserving compiler for ECC^D: ANF translation. The target language for this translation is ECC^A, a variant of ECC^D syntactically restricted to ANF. Recall from Chapter 3 that the first translation in the model compiler imposes a distinction between computation and values. A-normal form (ANF) translation accomplishes this goal and can take the place of CPS translation in the model compiler. (In fact, ANF was introduced as the essence of CPS (Flanagan et al., 1993).)

Digression. *The* A *in A-normal form has no further meaning. The* A *comes from the set of axioms and reductions introduced by Sabry and Felleisen (1992) to reason about CPS. Flanagan et al. (1993) use the* A *as a label in a commutative diagram, observing that there should exist a direct translation that produces a normal form equivalent to the CPS/optimization/un-CPS process found in practice. The programs produced by this translation are in normal form with respect to the* A*-reductions of Sabry and Felleisen,* i.e.*, in* A*-normal form.*

I start by explaining the key problems that arise when preserving dependent types through ANF translation and the main idea to my solution. I then move on to the formal development of the target language, the translation, and proofs of type preservation and compiler correctness. I conclude with a discussion of related problems, and a brief comparison of CPS and ANF in the context of dependent-type preservation, but leave a full discussion of CPS until Chapter 6.

Typographical Note. *In this chapter, I define fixed source and target languages. I typeset the source language, ECC^D, in a* blue, non-bold, sans-serif font*, and the target language, ECC^A, in a* **bold, red, serif font***.*

4.1 Main Ideas

The idea behind ANF is to make control flow explicit in the syntax of a program so transfer of control can be expressed as a jump. ANF encodes *computation* (*e.g.*, reducing an expression to a value) as sequencing simple intermediate computations with *let* expressions. To reduce e to a value in a high-level language, we need to describe the evaluation order and control flow of each language primitive. For example, if e is an application $e_1\ e_2$ and we want a call-by-value semantics, we say the language first evaluates e_1 to a value, then evaluates e_2 to a value, then performs the function application. ANF makes this explicit in the syntax by decomposing e into a series of primitive computations sequenced by **let**, as in $\mathbf{let}\,\mathbf{x}_0 = \mathbf{N}_0, ..., \mathbf{x}_n = \mathbf{N}_n\,\mathbf{in}\,\mathbf{N}$, where each $\mathbf{N}_i$ is either a value or a primitive computation applied to a value. The ANF translation of $e_1\ e_2$ is the following.

$$\begin{aligned}&\mathbf{let}\,\mathbf{x}_{1_0} = \mathbf{N}_{1_0}, ..., \mathbf{x}_{1_n} = \mathbf{N}_{1_n}, \mathbf{x}_1 = \mathbf{N}_1\\ &\quad\ \mathbf{x}_{2_0} = \mathbf{N}_{2_0}, ..., \mathbf{x}_{2_n} = \mathbf{N}_{2_n}, \mathbf{x}_2 = \mathbf{N}_2\\ &\quad\ \mathbf{in}\,(\mathbf{x}_1\ \mathbf{x}_2)\end{aligned}$$

$$\begin{aligned}\text{where } [\![e_1]\!] &= (\mathbf{let}\,\mathbf{x}_{1_0} = \mathbf{N}_{1_0}, ..., \mathbf{x}_{1_n} = \mathbf{N}_{1_n}\,\mathbf{in}\,\mathbf{N}_1)\\ [\![e_2]\!] &= (\mathbf{let}\,\mathbf{x}_{2_0} = \mathbf{N}_{2_0}, ..., \mathbf{x}_{2_n} = \mathbf{N}_{2_n}\,\mathbf{in}\,\mathbf{N}_2)\end{aligned}$$

Roughly, we can think of this translation as $\mathbf{let}\,\mathbf{x}_1 = [\![e_1]\!], \mathbf{x}_2 = [\![e_2]\!]\,\mathbf{in}\,\mathbf{x}_1\ \mathbf{x}_2$. This rough translation is only ANF if e_1 and e_2 are values or primitive computations. In general, the ANF translation reassociates all the intermediate computations from $[\![e_1]\!]$ and $[\![e_2]\!]$ so there are no nested **let** expressions. Once in ANF, it is simple to formalize a machine semantics to implement evaluation by always reducing the left-most computation, which will be a primitive operation.

The problem in developing a type-preserving translation for a dependently typed language is that changing the structure of a program disrupts the dependencies described in Chapter 2, *i.e.*, an expression e' whose type and evaluation depends on a sub-expression e. Recall that I call to a sub-expression such as e *depended upon*. This pattern happens in dependent elimination forms, such as application and projection. For example, for a dependent function $e_1 : \Pi\, x : A.\, B$, the application is typed as $e_1\ e_2 : B[e_2/x]$. Notice that the *depended upon* sub-expression e_2 is copied into the type. If we transform the expression $e_1\ e_2$, we can easily change the type. For example, recall the ANF translation for this expression given earlier.

$$\begin{aligned}&\mathbf{let}\,\mathbf{x}_{1_0} = \mathbf{N}_{1_0}, ..., \mathbf{x}_{1_n} = \mathbf{N}_{1_n}, \mathbf{x}_1 = \mathbf{N}_1\\ &\quad\ \mathbf{x}_{2_0} = \mathbf{N}_{2_0}, ..., \mathbf{x}_{2_n} = \mathbf{N}_{2_n}, \mathbf{x}_2 = \mathbf{N}_2\\ &\quad\ \mathbf{in}\,(\mathbf{x}_1\ \mathbf{x}_2)\end{aligned}$$

Using the standard typing rule for dependent **let** (without definitions), the type of the ANF translation above is $[\![B]\!][\mathbf{x}_2/\mathbf{x}]$ (the type of the body), with all **let**-bindings substituted into this type, *i.e.*, roughly $[\![B]\!][\mathbf{x}_2/\mathbf{x}][\mathbf{N}_{2_0}/\mathbf{x}_{2_0}][...][\mathbf{N}_{2_n}/\mathbf{x}_{2_n}][\mathbf{N}_2/\mathbf{x}_2]$. To show type preservation, we must show that the target type system can prove that this type is equivalent to the translation of the original type, *i.e.*, $[\![B[e_2/x]]\!]$. Intuitively, it *ought* to be true that the intermediate computations $(\mathbf{N}_{2_0}, ..., \mathbf{N}_{2_n}, \mathbf{N}_2)$ are equivalent to $[\![e_2]\!]$, since those are the computations that must happen to compute the value of e_2. Therefore, the two types *ought* to be equivalent.

But the intuitive argument that ANF ought to be type preserving is wrong. Instead of the $(\mathbf{x}_1\ \mathbf{x}_2)$ appearing in the body directly as above, consider the expression below in which $(\mathbf{x}_1\ \mathbf{x}_2)$ is bound and then used.

$$\mathbf{let}\,...\,\mathbf{in}\,\mathbf{let}\,\mathbf{y} = (\mathbf{x}_1\ \mathbf{x}_2)\,\mathbf{in}\,\mathbf{f}\ \mathbf{y} \qquad\qquad \text{where } \mathbf{f} : ([\![B]\!][[\![e_2]\!]/\mathbf{x}]) \to \mathbf{C}$$

To show type preservation, we now need to reestablish a dependent type that is **let**-bound, instead of in the body of a **let**. The type derivation fails, as follows.

$$\dfrac{\dfrac{}{\ldots \vdash (\mathbf{x}_1\ \mathbf{x}_2) : [\![\mathsf{B}]\!][\mathbf{x}_2/\mathbf{x}]} \qquad \dfrac{\text{fails}}{\ldots, \mathbf{y} : [\![\mathsf{B}]\!][\mathbf{x}_2/\mathbf{x}] \vdash \mathbf{f}\ \mathbf{y} : \mathbf{C}}}{\ldots \vdash \mathbf{let}\,\mathbf{y} = (\mathbf{x}_1\ \mathbf{x}_2)\,\mathbf{in}\,\mathbf{f}\ \mathbf{y} : \mathbf{C}[(\mathbf{x}_1\ \mathbf{x}_2)/\mathbf{y}]}$$

This fails since $\mathbf{f}$ expects $\mathbf{y} : ([\![\mathsf{B}]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}])$, but is applied to $\mathbf{y} : [\![\mathsf{B}]\!][\mathbf{x}_2/\mathbf{x}]$. We cannot show the two types are equal without substituting all the (elided) bindings as described earlier, but that substitution happens after it is needed. The problem is that the dependent typing rule for **let** only binds a depended-upon expression in the type of the *body* of the **let**, not in the types of *bound expressions*.

The typing rule for **let** only allows dependency in the type of the overall expression—that is, in the "output" of the typing judgment—but we need some way to thread dependencies into the sub-derivations while checking bound expressions. For example, if we could express dependencies in the *input*, something like $\ldots\mathbf{x}_2 = [\![\mathsf{e}_2]\!] \vdash (\mathbf{x}_1\ \mathbf{x}_2) : [\![\mathsf{B}]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}]$, then we could complete the derivation and prove type-preservation. And this is exactly the intuition we formalize.

We formalize this intuition using *definitions* (Severi and Poll, 1994) introduced in Chapter 2. Recall that the typing rule for **let** with definitions is the following.

$$\dfrac{\boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{A} \qquad \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} \vdash \mathbf{e}' : \mathbf{A}'}{\boldsymbol{\Gamma} \vdash \mathbf{let}\,\mathbf{x} = \mathbf{e}\,\mathbf{in}\,\mathbf{e}' : \mathbf{A}'[\mathbf{e}/\mathbf{x}]}$$

The definition $\mathbf{x} = \mathbf{e}$ is introduced when type checking the body of the **let**, and can be used to solve type equivalence in sub-derivations, instead of only in the substitution $\mathbf{A}'[\mathbf{e}/\mathbf{x}]$ in the "output" of the typing rule. While this is an extension to the type theory (so one may worry that the ANF translation applies to fewer dependently typed languages), 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.

Using definitions, we can prove that, under the definitions $\mathbf{x}_{2_0} = \mathbf{N}_{2_0}, \ldots, \mathbf{x}_{2_n} = \mathbf{N}_{2_n}, \mathbf{x}_2 = \mathbf{N}_2$ produced from the ANF translation $[\![\mathsf{e}_2]\!]$ (written `defs`$([\![\mathsf{e}_2]\!])$), the desired equivalence $\mathbf{x}_2 \equiv [\![\mathsf{e}_2]\!]$ holds. We use `hole`$([\![\mathsf{e}_2]\!])$ to refer to the innermost computation produced by the ANF translation, in this case `hole`$([\![\mathsf{e}_2]\!]) = \mathbf{N}_2$. Then we can prove `defs`$([\![\mathsf{e}_2]\!]) \vdash$ `hole`$([\![\mathsf{e}_2]\!]) \equiv [\![\mathsf{e}_2]\!]$ (formally captured by Lemma 4.2.5). Since we also have the definition, $\mathbf{x}_2 = \mathbf{N}_2$, we conclude that `defs`$([\![\mathsf{e}_2]\!]) \vdash \mathbf{x}_2 \equiv [\![\mathsf{e}_2]\!]$.

To formalize this type preservation argument, we need to step back and define the ANF translation precisely. In the source, looking at an expression such as $\mathsf{e}_1\ \mathsf{e}_2$, we do not know whether the expression is embedded in a larger context. This matters in ANF, since we can no longer compose expressions by nesting, but instead must compose expressions with **let**. This is why the above example translation disassembled the translation of e_1 and e_2 into a new, larger, **let** expression. To formalize the ANF translation, it helps to have a more compositional syntax for translating an expression and composing it with an unknown context.

I develop a compositional translation by indexing 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. A continuation $\mathbf{K}$ is a program with a hole (single linear variable) $[\cdot]$, and can be composed with a computation $\mathbf{K}[\mathbf{N}]$ to form a program $\mathbf{M}$. 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 functions and application as follows.

$$\llbracket \lambda\, x : A.\, e \rrbracket\ \mathbf{K} = \mathbf{K}[\boldsymbol{\lambda}\, \mathbf{x} : (\llbracket A \rrbracket\ [\cdot]).\ (\llbracket e \rrbracket\ [\cdot])]$$
$$\llbracket e_1\ e_2 \rrbracket\ \mathbf{K} = \llbracket e_1 \rrbracket\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \llbracket e_2 \rrbracket\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]$$

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

The key typing rule for continuations is the following.

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

The type $(\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ of continuations describes that the continuation must be composed with the term $\mathbf{N}$ of type $\mathbf{A}$, and the result will be of type $\mathbf{B}$. This expresses syntactically the intuition that continuations must be used linearly to avoid control effects, which are known to cause inconsistency with dependent types (Barthe and Uustalu, 2002; Herbelin, 2005). Note that this type allows us to introduce the definition $\mathbf{y} = \mathbf{N}$ via the type, before we know how the continuation is used.[1] I discuss this rule further in Section 4.2. The Lemma 4.2.4 (Cut) expresses that continuation typing is not an extension to the target type theory, which is important to ensure ANF can be applied in practice.

The key lemma to prove type preservation is the following.

Lemma 4.1.1. *If* $\Gamma \vdash e : A$ *and* $\boldsymbol{\Gamma}, \texttt{defs}(\llbracket e \rrbracket) \vdash \mathbf{K} : (\texttt{hole}(\llbracket e \rrbracket) : \llbracket A \rrbracket) \Rightarrow \mathbf{B}$*, then* $\boldsymbol{\Gamma} \vdash \llbracket e \rrbracket\ \mathbf{K} : \mathbf{B}$.

This lemma captures the fact that each time we build a new $\mathbf{K}$ in the ANF translation, we must show it is well-typed, and that is where we apply the reasoning about definitions. Proving that $\mathbf{K}$ has the above type requires proving $\boldsymbol{\Gamma}, \texttt{defs}(\llbracket e \rrbracket) \vdash \texttt{hole}(\llbracket e \rrbracket) : \llbracket A \rrbracket$. For our running example, this means proving $\boldsymbol{\Gamma}, \texttt{defs}(\llbracket e_1\ e_2 \rrbracket) \vdash \mathbf{x}_1\ \mathbf{x}_2 : \llbracket B[e_2/x] \rrbracket$. Recall that $\mathbf{x}_1\ \mathbf{x}_2 : \llbracket B \rrbracket[\mathbf{x}_2/\mathbf{x}]$. As we saw earlier, definitions allow us to prove that $\boldsymbol{\Gamma}, \texttt{defs}(\llbracket e_1\ e_2 \rrbracket) \vdash \mathbf{x}_2 \equiv \llbracket e_2 \rrbracket$; therefore, combined with compositionality ($\llbracket B[e_2/x] \rrbracket \equiv \llbracket B \rrbracket[\llbracket e_2 \rrbracket/\mathbf{x}]$, Lemma 4.3.3), the proof is complete! The earlier failing derivation now succeeds:

1 This is essentially a singleton type, but as mentioned in Chapter 2, I avoid explicit encoding with singleton types to focus on the intuition: tracking dependencies.

$$
\begin{array}{llcl}
\textit{Universes} & \mathbf{U} & ::= & \mathbf{Prop} \mid \mathbf{Type}_i \\
\textit{Values} & \mathbf{V} & ::= & \mathbf{x} \mid \mathbf{U} \mid \Pi\,\mathbf{x} : \mathbf{M}.\,\mathbf{M} \mid \lambda\,\mathbf{x} : \mathbf{M}.\,\mathbf{M} \mid \Sigma\,\mathbf{x} : \mathbf{M}.\,\mathbf{M} \\
 & & \mid & \langle \mathbf{V}, \mathbf{V} \rangle\ \mathbf{as}\ \mathbf{M} \\
\textit{Computations} & \mathbf{N} & ::= & \mathbf{V} \mid \mathbf{V}\ \mathbf{V} \mid \mathbf{fst}\ \mathbf{V} \mid \mathbf{snd}\ \mathbf{V} \\
\textit{Configurations} & \mathbf{M}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{N} \mid \mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{M} \\
\textit{Continuations} & \mathbf{K} & ::= & [\cdot] \mid \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{M}
\end{array}
$$

Figure 4.1: ECC^A Syntax

$$
\dfrac{\dfrac{\textit{Lemma } 4.3.3 \qquad \textit{Lemma } 4.2.5}{\mathtt{defs}([\![e_1\ e_2]\!]) \vdash (\mathbf{x}_1\ \mathbf{x}_2) : [\![B]\!][[\![e_2]\!]/\mathbf{x}]} \qquad \dfrac{\text{succeeds!}}{\mathbf{y} : [\![B]\!][[\![e_2]\!]/\mathbf{x}] \vdash \mathbf{f}\ \mathbf{y} : \mathbf{C}}}{\mathtt{defs}([\![e_1\ e_2]\!]) \vdash \mathbf{let}\ \mathbf{y} = (\mathbf{x}_1\ \mathbf{x}_2)\ \mathbf{in}\ \mathbf{f}\ \mathbf{y} : \mathbf{C}[(\mathbf{x}_1\ \mathbf{x}_2)/\mathbf{y}]}
$$

4.2 ANF Intermediate Language

The target language, ECC^A, is an ANF-restricted subset of ECC^D. For now, I exclude dependent conditionals from ECC^D and from ECC^A; I return to them in Section 4.4. I continue to use the same typing and conversion rules as ECC^D, which are permitted to break ANF when computing term equivalence during type checking. However, I define an ANF-preserving machine-like semantics for evaluation of program configurations. Note that this means the definitional equivalence is not suitable for equational reasoning about run-time terms (*e.g.*, reasoning about optimizations), without ANF translation afterwards.[2]

Typographical Note. *Although ECC^A is a restriction of ECC^D, I typeset it as separate language for clarity, and use the shift in fonts to indicate an explicit shift in how I am treating terms,* i.e., *as either ANF-restricted terms still suitable for evaluation, or as unrestricted terms that we can type check but cannot run in the ANF semantics any longer.*

I give the syntax for ECC^A in Figure 4.1. As discussed in Chapter 3, the goal is to make an explicit distinction between values and computations. In ECC^A, I do this by imposing a syntactic distinction between *values* $\mathbf{V}$ which do not reduce, *computations* $\mathbf{N}$

2 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 is meant to support ANF transformation and optimization with join points.

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

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

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

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

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

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

Figure 4.2: ECC^A Composition of Configurations

which eliminate values and can be composed using *continuations* $\mathbf{K}$, and *configurations* $\mathbf{M}$ which intuitively represent whole programs ready to be executed. A continuation $\mathbf{K}$ is a program with a hole, and is composed $\mathbf{K}[\mathbf{N}]$ with a computation $\mathbf{N}$ to form a configuration $\mathbf{M}$. For example, $(\mathbf{let}\,\mathbf{x} = [\cdot]\,\mathbf{in}\,\mathbf{snd}\,\mathbf{x})[\mathbf{N}] = \mathbf{let}\,\mathbf{x} = \mathbf{N}\,\mathbf{in}\,\mathbf{snd}\,\mathbf{x}$. Since continuations are not first-class objects in the language, we cannot express control effects—continuations are syntactically guaranteed to be used linearly. [3] Note that despite the syntactic distinction, I still do not enforce a phase distinction—configurations (programs) can appear in types.

In ANF, all continuations are left associated, so substitution can break ANF. Note that β-reduction takes an ANF configuration $\mathbf{K}[(\lambda\,\mathbf{x} : \mathbf{A}.\,\mathbf{M})\ \mathbf{V}]$ but would naïvely produce $\mathbf{K}[\mathbf{M}[\mathbf{V}/\mathbf{x}]]$. While the substitution $\mathbf{M}[\mathbf{V}/\mathbf{x}]$ is well-defined, substituting the resulting term, itself a *configuration*, into the continuation $\mathbf{K}$ could result in the non-ANF term $\mathbf{let}\,\mathbf{x} = \mathbf{M}\,\mathbf{in}\,\mathbf{M}'$. In ANF, configurations cannot be nested.

To ensure reduction preserves ANF, I define composition of a continuation $\mathbf{K}$ and a configuration $\mathbf{M}$, Figure 4.2, typically called *renormalization* in the literature (Sabry and Wadler, 1997; Kennedy, 2007). When composing a continuation with a configuration, $\mathbf{K}\langle\langle \mathbf{M} \rangle\rangle$, we essentially unnest all continuations so they remain left-associated.[4] Note that these definitions are simplified because I am ignoring capture-avoiding substitution.

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

3 The reader familiar with proof theory may wish to consider configurations and continuations as the same term under a *stoup* (Girard, 1991), an environment expressing either zero or one linear computation variables $[\cdot]$; when the stoup is empty, we have a configuration and otherwise we have a continuation.
4 Some work uses an append notation, *e.g.*, $\mathbf{M} :: \mathbf{K}$ (Sabry and Wadler, 1997), suggesting we are appending $\mathbf{K}$ onto the continuation for $\mathbf{M}$; I prefer notation that evokes composition.

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

$$\begin{array}{rcl}
\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M})\ \mathbf{V}] & \mapsto_{\beta} & \mathbf{K}\langle\langle \mathbf{M}[\mathbf{V}/\mathbf{x}] \rangle\rangle \\
\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] & \mapsto_{\pi_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{snd}\ \langle \mathbf{V}_1, \mathbf{V}_2 \rangle] & \mapsto_{\pi_2} & \mathbf{K}[\mathbf{V}_2] \\
\mathbf{let}\, \mathbf{x} = \mathbf{V}\, \mathbf{in}\, \mathbf{M} & \mapsto_{\zeta} & \mathbf{M}[\mathbf{V}/\mathbf{x}]
\end{array}$$

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

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

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

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

Figure 4.3: ECC^A Evaluation

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 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* (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.

I present the call-by-value (CBV) evaluation semantics for ECC^A in Figure 4.3. It is essentially standard, but recall that β-reduction produces a configuration $\mathbf{M}$ which must be composed with the current continuation $\mathbf{K}$. This semantics is only for the evaluation of configurations; during type checking, we continue to use the type system and conversion relation defined in Chapter 2.

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

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

Figure 4.4: ECC^A Continuation Typing

4.2.1 The Essence of Dependent Continuation Typing

I define continuation typing in Figure 4.4. The type $(\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ of a continuation expresses that this continuation expects to be composed with a term equal (syntactically) to the computation $\mathbf{N}$ of type $\mathbf{A}$ and returns a result of type $\mathbf{B}$ when completed. This is the formal statement that $\mathbf{N}$ is depended upon (in the sense introduced in Chapter 2) in the rest of the computation, and is key to recovering the dependency disrupted during ANF translation. For the empty continuation $[\cdot]$, $\mathbf{N}$ is arbitrary since an empty continuation has no "rest of the program" that could depend on anything.

Intuitively, what we want from continuation typing is a compositionality property—that we can reason about the types of configurations $\mathbf{K}[\mathbf{N}]$ by composing the typing derivations for $\mathbf{K}$ and $\mathbf{N}$. To get this property, a continuation type must express not merely the *type* of its hole $\mathbf{A}$, but exactly *which term* $\mathbf{N}$ will be bound in the hole. We see this formally from the typing rule Rule Let (the same for ECC^A as for ECC^D in Chapter 2), since showing that $\mathbf{let}\, \mathbf{y} = \mathbf{N}\, \mathbf{in}\, \mathbf{M}$ is well-typed requires showing that $\mathbf{y} = \mathbf{N} \vdash \mathbf{M}$, that is, requires knowing the definition $\mathbf{y} = \mathbf{N}$. If we omit the expression $\mathbf{N}$ from the type of continuations, 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* $\mathbf{y}$. I prove that continuation typing is compositional in this way, Lemma 4.2.4 (Cut).

Note that the type of the result in a continuation type cannot depend on the term that will be plugged in for the hole, *i.e.*, for a continuation $\mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$, $\mathbf{B}$ does not depend on $\mathbf{N}$. This is not important for ANF translation, but is interesting as it provides insight into related work as I discuss in Section 4.4. The restriction is not necessary, and is not true in all systems, but turns out to be true in ANF. To see this, first note that the initial continuation must be empty and thus *cannot* have a result type that depends on its hole. The ANF translation will take this initial empty continuation and compose it with intermediate continuations $\mathbf{K}'$. Since composing any continuation $\mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ 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. This is similar to how, in CPS, the answer type doesn't matter and might as well be $\bot$.

4.2.2 Meta-Theory

Since ECC^A is a syntactic discipline in ECC^D, we inherit most of the meta-theory from ECC^D, notably: logical consistency, type safety, and decidability (Luo, 1990; Severi and Poll, 1994). There are some new meta-theoretic questions to answer, though, such as: Is ANF evaluation sound? Does continuation typing make sense?

First, I prove that ANF evaluation semantics is sound with respect to definitional equivalence. That is, running in the ANF evaluation semantics produces an equivalent value to normalization in the equivalence relation. The heart of this proof is actually *naturality*, a property found in the literature on continuations that essentially expressed freedom from control effects (Thielecke, 2003).

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

First, I prove that composing continuation in ANF is sound with respect to substitution. This is an expression 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 $\mathbf{K}$.

Lemma 4.2.1 (Naturality). $\mathbf{K}\langle\langle\mathbf{M}\rangle\rangle \equiv \mathcal{ST}(\mathbf{K}[\mathbf{M}])$

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

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

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

Must show that $\mathbf{let}\ \mathbf{x} = \mathbf{N'}\ \mathbf{in}\ \mathbf{K}\langle\langle\mathbf{M'}\rangle\rangle \equiv \mathcal{ST}(\mathbf{K}[\mathbf{let}\ \mathbf{x} = \mathbf{N'}\ \mathbf{in}\ \mathbf{M}])$. Note that this requires breaking ANF while computing equivalence.

$$\begin{aligned}
& \mathbf{let}\ \mathbf{x} = \mathbf{N'}\ \mathbf{in}\ \mathbf{K}\langle\langle\mathbf{M'}\rangle\rangle \\
\triangleright_\zeta\ & \mathcal{ST}(\mathbf{K}\langle\langle\mathbf{M'}\rangle\rangle[\mathbf{N'}/\mathbf{x}]) && (1) \\
& \text{note that this substitution is undefined in ANF} \\
=\ & \mathbf{K}\langle\langle\mathcal{ST}(\mathbf{M'}[\mathbf{N'}/\mathbf{x}])\rangle\rangle && (2) \\
& \text{by ignoring capture-avoiding substitution} \\
\triangleleft^*\ & \mathcal{ST}(\mathbf{K}[\mathbf{let}\ \mathbf{x} = \mathbf{N'}\ \mathbf{in}\ \mathbf{M}]) && (3) \\
& \text{by } \zeta\text{-reduction and congruence}
\end{aligned}$$

□

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

Lemma 4.2.2 (Small-step soundness). *If* $\mathbf{M} \mapsto \mathbf{M}'$ *then* $\vdash \mathbf{M} \equiv \mathbf{M}'$

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

Case: $\mathbf{K}[(\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{A}.\,\mathbf{M}_1)\ \mathbf{V}] \mapsto_\beta \mathbf{K}\langle\langle \mathbf{M}_1[\mathbf{V}/\mathbf{x}]\rangle\rangle$

Must show that $\mathbf{K}[(\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{A}.\,\mathbf{M}_1)\ \mathbf{V}] \equiv \mathbf{K}\langle\langle \mathbf{M}_1[\mathbf{V}/\mathbf{x}]\rangle\rangle$

$$\begin{aligned}
& \mathbf{K}[(\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{A}.\,\mathbf{M}_1)\ \mathbf{V}] && \\
\rhd^* \;& \mathcal{ST}(\mathbf{K}[\mathbf{M}_1[\mathbf{V}/\mathbf{x}]]) && \text{by } \beta \text{ and congruence} \qquad (4)\\
\equiv \;& \mathbf{K}\langle\langle \mathbf{M}_1[\mathbf{V}/\mathbf{x}]\rangle\rangle && \text{by Lemma 4.2.1} \qquad (5)
\end{aligned}$$

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

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

Case: $n = 0$ By Rule Red-Refl and Rule $\equiv$.

Case: $n = i + 1$ Follows by Lemma 4.2.2 and the induction hypothesis. □

I prove that plugging a well-typed term into a well-typed continuation results in a well-typed term of the expected type. This theorem corresponds to the Rule Cut rule from sequent calculus, and tells us that continuation typing allows for compositional reasoning about configurations $\mathbf{K}[\mathbf{N}]$ whose result types do not depend on $\mathbf{N}$.

Lemma 4.2.4 (Cut). *If* $\boldsymbol{\Gamma} \vdash \mathbf{K} : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{B}$ *and* $\boldsymbol{\Gamma} \vdash \mathbf{N} : \mathbf{A}$ *then* $\boldsymbol{\Gamma} \vdash \mathbf{K}[\mathbf{N}] : \mathbf{B}$.

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

Case: $\boldsymbol{\Gamma} \vdash [\cdot] : (\mathbf{N} : \mathbf{A}) \Rightarrow \mathbf{A}$, trivial

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

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

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

$$\texttt{defs}(\mathbf{M}) = \mathbf{x}_1 = \mathbf{N}_1, \dots, \mathbf{x}_n = \mathbf{N}_n \quad \text{where}\, \mathbf{M} = \mathbf{let}\, \mathbf{x}_1 = \mathbf{N}_1\, \mathbf{in}\, \dots \mathbf{let}\, \mathbf{x}_n = \mathbf{N}_n\, \mathbf{in}\, \mathbf{N}_{n+1}$$

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

$$\texttt{hole}(\mathbf{M}) = \mathbf{N}_{n+1} \qquad \text{where } \mathbf{M} = \mathbf{let}\, \mathbf{x}_1 = \mathbf{N}_1\, \mathbf{in}\, \dots \mathbf{let}\, \mathbf{x}_n = \mathbf{N}_n\, \mathbf{in}\, \mathbf{N}_{n+1}$$

Figure 4.5: ECC^A Continuation Exports

To reason inductively about ANF terms, we need to separate a configuration $\mathbf{M}$ into its exported definitions $\texttt{defs}(\llbracket \mathbf{M} \rrbracket)$ and its underlying computation $\texttt{hole}(\llbracket \mathbf{M} \rrbracket)$. In Figure 4.5, I define $\texttt{defs}(\llbracket \mathbf{M} \rrbracket)$ to be the definitions exported by the ANF term $\mathbf{M}$. These are the definitions that will be in scope for a continuation $\mathbf{K}$ when composed with $\mathbf{M}$, *i.e.*, in scope for $\mathbf{K}$ in $\mathbf{K}\langle\langle \mathbf{M} \rangle\rangle$. I define this as $\texttt{hole}(\llbracket \mathbf{M} \rrbracket)$, also in Figure 4.5. Note that $\texttt{hole}(\llbracket \mathbf{M} \rrbracket)$ will only be well typed in the environment for $\mathbf{M}$ extended with the definitions $\texttt{defs}(\llbracket \mathbf{M} \rrbracket)$.

I show that a configuration is nothing more than its exported definitions and underlying computation, *i.e.*, that in a context with the exports of $\texttt{defs}(\llbracket \mathbf{M} \rrbracket)$, $\texttt{hole}(\llbracket \mathbf{M} \rrbracket) \equiv \mathbf{M}$. In essence, this lemma shows how ANF converts a dependency on a *configuration* $\mathbf{M}$ into a series of dependencies on *values*, *i.e.*, the names $\mathbf{x}_0, \dots, \mathbf{x}_{n+1}$ in $\texttt{defs}(\llbracket \mathbf{M} \rrbracket)$. Note that the ANF guarantees that all dependent typing rules, like $\mathbf{V}\ \mathbf{V}' : \mathbf{B}[\mathbf{V}'/\mathbf{x}]$, only depend on values. This lemma allows us to recover the dependency on a configuration.

Lemma 4.2.5. $\texttt{defs}(\mathbf{M}) \vdash \texttt{hole}(\mathbf{M}) \equiv \mathbf{M}$

Proof. Note that the exports $\texttt{defs}(\mathbf{M})$ are exactly the definitions from the syntax of $\mathbf{M}$. Inlining those definitions via δ-reduction is the same as reducing $\mathbf{M}$ via ζ-reduction.

$$\begin{aligned} \mathbf{M} = &\ (\mathbf{let}\, \mathbf{x}_1 = \mathbf{N}_1\, \mathbf{in}\, \dots \mathbf{let}\, \mathbf{x}_n = \mathbf{N}_n\, \mathbf{in}\, \mathbf{N}_{n+1}) && (6) \\ \triangleright_\zeta^n &\ \mathbf{N}_{n+1}[\mathbf{N}_1 \dots \mathbf{N}_n/\mathbf{x}_1 \dots \mathbf{x}_n] && (7) \end{aligned}$$

And $\texttt{hole}(\mathbf{M}) = \mathbf{N}_{n+1} \triangleright_\delta^n \mathbf{N}_{n+1}[\mathbf{N}_1 \dots \mathbf{N}_n/\mathbf{x}_1 \dots \mathbf{x}_n]$ □

4.3 ANF Translation

The ANF translation is presented in Figure 4.6. The translation is defined inductively on the syntax of the source term, and is indexed by a current continuation. The translation is essentially standard. When translating a value such as x, $\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}$, and Type_i, we essentially plug the value into the current continuation, recursively translating the sub-expressions of the value if applicable. For non-values such as application, we make sequencing explicit by recursively translating each sub-expression with a continuation that binds the result of the sub-expression and will perform the rest of the computation.

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

$$
\begin{aligned}
[\![e]\!] &\overset{\text{def}}{=} [\![e]\!]\ [\cdot] \\
[\![x]\!]\ \mathbf{K} &\overset{\text{def}}{=} \mathbf{K}[\mathbf{x}] \\
[\![\mathsf{Prop}]\!]\ \mathbf{K} &\overset{\text{def}}{=} \mathbf{K}[\mathbf{Prop}] \\
[\![\mathsf{Type}_i]\!]\ \mathbf{K} &\overset{\text{def}}{=} \mathbf{K}[\mathbf{Type}_i] \\
[\![\Pi\, x : A.\, B]\!]\ \mathbf{K} &\overset{\text{def}}{=} \mathbf{K}[\mathbf{\Pi\, x} : [\![A]\!].\, [\![B]\!]] \\
[\![\lambda\, x : A.\, e]\!]\ \mathbf{K} &\overset{\text{def}}{=} \mathbf{K}[\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e]\!]] \\
[\![e_1\ e_2]\!]\ \mathbf{K} &\overset{\text{def}}{=} [\![e_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ ([\![e_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]) \\
[\![\Sigma\, x : A.\, B]\!]\ \mathbf{K} &\overset{\text{def}}{=} \mathbf{K}[\mathbf{\Sigma\, x} : [\![A]\!].\, [\![B]\!]] \\
[\![\langle e_1, e_2\rangle\ \mathsf{as}\ A]\!]\ \mathbf{K} &\overset{\text{def}}{=} [\![e_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[(\langle \mathbf{x}_1, \mathbf{x}_2\rangle\ \mathbf{as}\ [\![A]\!])]) \\
[\![\mathsf{fst}\ e]\!]\ \mathbf{K} &\overset{\text{def}}{=} [\![e]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{fst}\ \mathbf{x}] \\
[\![\mathsf{snd}\ e]\!]\ \mathbf{K} &\overset{\text{def}}{=} [\![e]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x}] \\
[\![\mathsf{let}\ x = e\ \mathsf{in}\ e']\!]\ \mathbf{K} &\overset{\text{def}}{=} [\![e]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ [\![e']\!]\ \mathbf{K}
\end{aligned}
$$

Figure 4.6: ANF Translation from ECC^D to ECC^A

Note that if the translation must produce type annotations for input to a continuation, then defining the translation and typing preservation proof are somewhat more complicated. For instance, if we required the **let**-bindings in the target language to have type annotations for bound expressions, then we would need to modify the translation to produce those annotations. This requires defining the translation over typing derivations, so the compiler has access to the type of the expression and not only its syntax. I discuss the implications of this in Section 4.4.

Next, I show type preservation following essentially the standard architecture presented Chapter 3. A few additional lemmas are required, and some lemma statements are non-standard, as I discuss next.

After proving type preservation, I prove correctness of separate compilation for the ANF machine semantics. I use the same notion of linking and observations as defined in Chapter 2. This proof is straightforward from the meta-theory about the machine semantics proved in Section 4.2, and from equivalence preservation.

4.3.1 Type Preservation

To prove type preservation, I follow the same recipe as presented in Chapter 3. First, I show compositionality, which states that the translation commutes with substitution, *i.e.*, that substituting first and then translating is equivalent to translating first and then substituting. This proof is somewhat non-standard for ANF since the notion of composition in ANF is not the usual substitution. Next, I show that reduction and conversion are preserved up to equivalence, as is standard. Note that for this theorem, we are interested in the conversion semantics used for definitional equivalence, not in the machine semantics. Then, I show equivalence preservation: if two terms are definitionally equivalent in the source, then their translations are definitionally equivalent. Finally, I can show type preservation of the ANF translation, using continuation typing to express the inductive invariant required for ANF. The lemma for type preservation is non-standard compared to the generic statement given in Chapter 3. The continuation typing allows us to formally state type preservation in terms of the intuitive reason it should be true: because the definitions expressed by the continuation typing suffice to prove equivalence between a computation variable and the original depended-upon expression.

Typographical Note. *Recall from Section 4.2, I shift from the* ***target language font*** *to the source language font whenever the term is no longer in ANF, such as when performing standard substitution or conversion.*

Before I proceed, I state a property about the syntax produced by the ANF translation, in particular, that the ANF translation does produce syntax in ANF. The proof is straightforward so I elide it.

Theorem 4.3.1 (Normal Form). $[\![\mathsf{e}]\!]\ \mathbf{K'} = \mathbf{let\ x_1 = N_1\ in} \ldots \mathbf{let\ x_n = N_n\ in\ K'[N_{n+1}]}$

As discussed in Section 4.2, composition in ANF is somewhat non-standard. Normally, we compose via substitution and compositionality is $[\![\mathsf{e}[\mathsf{e'}/\mathsf{x}]]\!] \equiv [\![\mathsf{e}]\!][[\![\mathsf{e'}]\!]/\mathbf{x}]$, which says we can either compose then translate or translate then compose. However most composition in ANF goes through continuations, not through substitution, since only values can be substituted in ANF. The primary compositionality lemma (Lemma 4.3.2) tells us that 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. Note that this proof is entirely within ECC^A; there are no language boundaries.

Lemma 4.3.2 (Compositionality). $\mathbf{K'}\langle\!\langle\ [\![\mathsf{e}]\!]\ \mathbf{K}\rangle\!\rangle = [\![\mathsf{e}]\!]\ \mathbf{K'}\langle\!\langle\mathbf{K}\rangle\!\rangle$

Proof. By induction on the structure of e. All value cases are trivial. The cases for non-values are all essentially similar, by definition of composition for continuations or configurations. I 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 $\mathbf{K'}\langle\!\langle\mathbf{K}[\mathbf{\Pi\,x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!]]\rangle\!\rangle = \mathbf{K'}\langle\!\langle\mathbf{K}[\mathbf{\Pi\,x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!]]\rangle\!\rangle$, which is trivial. Note that we need not appeal to induction, since the recursive translation does not use the current continuation for values.

Case: $e = e_1\ e_2$ Must show that

$$\begin{aligned} & \mathbf{K}'\langle\langle(\llbracket e_1 \rrbracket\,(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,(\llbracket e_2 \rrbracket\,\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])))\rangle\rangle \\ = & (\llbracket e_1 \rrbracket\,(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,(\llbracket e_2 \rrbracket\,\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle[\mathbf{x}_1\ \mathbf{x}_2]))) \end{aligned}$$

The proof follows essentially from the definition of continuation composition.

$$\begin{aligned} & \mathbf{K}'\langle\langle(\llbracket e_1 \rrbracket\,(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,(\llbracket e_2 \rrbracket\,\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])))\rangle\rangle \\ = & (\llbracket e_1 \rrbracket\,\mathbf{K}'\langle\langle(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,(\llbracket e_2 \rrbracket\,\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]))\rangle\rangle) && (8) \\ & \text{by the induction hypothesis applied to } e_1 \\ = & (\llbracket e_1 \rrbracket\,(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,\mathbf{K}'\langle\langle(\llbracket e_2 \rrbracket\,\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2])\rangle\rangle)) && (9) \\ & \text{by definition of continuation composition} \\ = & (\llbracket e_1 \rrbracket\,(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,(\llbracket e_2 \rrbracket\,\mathbf{K}'\langle\langle\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]\rangle\rangle))) && (10) \\ & \text{by the induction hypothesis applied to } e_2 \\ = & (\llbracket e_1 \rrbracket\,(\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,(\llbracket e_2 \rrbracket\,\mathbf{let}\,\mathbf{x}_2 = [\cdot]\,\mathbf{in}\,\mathbf{K}'\langle\langle\mathbf{K}\rangle\rangle[\mathbf{x}_1\ \mathbf{x}_2]))) && (11) \\ & \text{by definition of continuation composition} && \square \end{aligned}$$

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

Lemma 4.3.3 (Substitution). $\llbracket e[e'/x] \rrbracket\ \mathbf{K} \equiv \mathcal{ST}((\llbracket e \rrbracket\ \mathbf{K})[\llbracket e' \rrbracket/\mathbf{x}])$

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

Case: $e = x$

Must show that $\llbracket e' \rrbracket\ \mathbf{K} \equiv \mathcal{ST}((\llbracket x \rrbracket\ \mathbf{K})[\llbracket e' \rrbracket/\mathbf{x}])$

$$\begin{aligned} & \mathcal{ST}(\llbracket x \rrbracket\ \mathbf{K}[\llbracket e' \rrbracket/\mathbf{x}]) \\ = & \mathcal{ST}(\mathbf{K}[\mathbf{x}][\llbracket e' \rrbracket/\mathbf{x}]) && && (12) \\ = & \mathcal{ST}(\mathbf{K}[\llbracket e' \rrbracket]) && && (13) \\ \equiv & \mathbf{K}\langle\langle\ \llbracket e' \rrbracket\ \rangle\rangle && \text{by Lemma 4.2.1} && (14) \\ \equiv & \ \llbracket e' \rrbracket\ \mathbf{K} && \text{by Lemma 4.3.2} && (15) \end{aligned}$$

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

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

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

$$
\begin{aligned}
& [\![\Pi\, x' : A.\, B[e'/x]]\!]\ \mathbf{K} \\
= \;& [\![\Pi\, x' : A[e'/x].\, B[e'/x]]\!]\ \mathbf{K} && (16)\\
= \;& \mathbf{K}[\mathbf{\Pi\, x'} : [\![A[e'/x]]\!].\ [\![B[e'/x]]\!]] && (17)\\
\equiv \;& \mathbf{K}[\mathbf{\Pi\, x'} : \mathcal{ST}([\![A]\!][[\![e']\!]/\mathbf{x}]).\, \mathcal{ST}([\![B]\!][[\![e']\!]/\mathbf{x}])] && (18)\\
& \text{by the induction hypothesis}\\
= \;& \mathcal{ST}(\mathbf{K}[\mathbf{\Pi\, x'} : [\![A]\!].\ [\![B]\!]][[\![e']\!]/\mathbf{x}]) && (19)\\
& \text{by definition of substitution}\\
= \;& \mathcal{ST}(([\![\Pi\, x' : A.\, B]\!]\ \mathbf{K})[[\![e']\!]/\mathbf{x}]) && (20)\\
& \text{by definition}
\end{aligned}
$$

Case: $e = e_1\ e_2$

Must show that $[\![(e_1\ e_2)[e'/\mathbf{x}]]\!]\ \mathbf{K} \equiv \mathcal{ST}(([\![e_1\ e_2]\!]\ \mathbf{K})[[\![e']\!]/\mathbf{x}])$

$$
\begin{aligned}
& [\![(e_1\ e_2)[e'/x]]\!]\ \mathbf{K} \\
= \;& [\![e_1[e'/x]\ e_2[e'/x]]\!]\ \mathbf{K} && (21)\\
& \text{by substitution}\\
= \;& [\![e_1[e'/x]]\!]\ \mathbf{let\, x_1} = [\cdot]\ \mathbf{in}\ [\![e_2[e'/x]]\!]\ \mathbf{let\, x_2} = [\cdot]\ \mathbf{in\, K}[\mathbf{x_1\ x_2}] && (22)\\
& \text{by translation}\\
\equiv \;& [\![e_1[e'/x]]\!]\ \mathbf{let\, x_1} = [\cdot]\ \mathbf{in}\, \mathcal{ST}(([\![e_2]\!]\ \mathbf{let\, x_2} = [\cdot]\ \mathbf{in\, K}[\mathbf{x_1\ x_2}])[[\![e']\!]/\mathbf{x}]) && (23)\\
& \text{by IH applied to } e_1\\
\equiv \;& [\![e_1]\!]\ \mathbf{let\, x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let\, x_2} = [\cdot]\ \mathbf{in\, K}[\mathbf{x_1\ x_2}][[\![e']\!]/x][[\![e']\!]/\mathbf{x}] && (24)\\
& \text{by IH applied to } e_2\\
= \;& \mathcal{ST}(([\![e_1]\!]\ \mathbf{let\, x_1} = [\cdot]\ \mathbf{in}\ [\![e_2]\!]\ \mathbf{let\, x_2} = [\cdot]\ \mathbf{in\, K}[\mathbf{x_1\ x_2}])[[\![e']\!]/\mathbf{x}]) && (25)\\
& \text{by substitution}\\
= \;& \mathcal{ST}(([\![e_1\ e_2]\!]\ \mathbf{K})[[\![e']\!]/\mathbf{x}]) && (26)\\
& \text{by substitution}
\end{aligned}
$$

□

Since equivalence is part of the type system, to show type preservation, we must show that equivalence is preserved. I first show that reduction is preserved up to equivalence, then conversion, and finally that equivalence is preserved. The proofs are straightforward; intuitively, ANF is just adding a bunch of ζ-reductions.

Lemma 4.3.4 (Preservation of Reduction)**.** *If* $\Gamma \vdash e \triangleright e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$.

Proof. By cases on $\Gamma \vdash e \triangleright e'$. I give the key cases.

Case: $\Gamma \vdash x \rhd_\delta e'$

We must show that $[\![\Gamma]\!] \vdash [\![x]\!] \equiv [\![e']\!]$

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

Case: $\Gamma \vdash \lambda x : A.\, e_1 \; e_2 \rhd_\beta e_1[e_2/x]$

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

$$\begin{aligned}
& [\![\lambda x : A.\, e_1 \; e_2]\!] \\
= \;& [\![\lambda x : A.\, e_1]\!] \; \mathbf{let}\; \mathbf{x}_1 = [\cdot] \;\mathbf{in}\; [\![e_2]\!] \;\mathbf{let}\; \mathbf{x}_2 = [\cdot] \;\mathbf{in}\; \mathbf{x}_1 \; \mathbf{x}_2 && (27)\\
= \;& \mathbf{let}\; \mathbf{x}_1 = (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!]) \;\mathbf{in}\; [\![e_2]\!] \;\mathbf{let}\; \mathbf{x}_2 = [\cdot] \;\mathbf{in}\; \mathbf{x}_1 \; \mathbf{x}_2 && (28)\\
\rhd^* \;& [\![e_2]\!] \;\mathbf{let}\; \mathbf{x}_2 = [\cdot] \;\mathbf{in}\; \boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!] \;\; \mathbf{x}_2 && (29)\\
= \;& \mathbf{let}\; \mathbf{x}_2 = [\cdot] \;\mathbf{in}\; (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!]) \; \mathbf{x}_2 \langle\!\langle\, [\![e_2]\!] \,\rangle\!\rangle && \text{(Lemma 4.3.2)} \quad (30)\\
\equiv \;& \mathsf{let}\; x_2 = [\![e_2]\!] \;\mathsf{in}\; (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!]) \; x_2 && \text{(Lemma 4.2.1)} \quad (31)\\
\rhd_\zeta \;& (\boldsymbol{\lambda}\, \mathbf{x} : [\![A]\!].\, [\![e_1]\!]) \;\; [\![e_2]\!] && (32)\\
\rhd_\beta \;& \mathcal{ST}([\![e_1]\!][[\![e_2]\!]/\mathbf{x}]) && (33)\\
\equiv \;& [\![e_1[e_2/x]]\!] && \text{(Lemma 4.3.3)} \quad (34)
\end{aligned}$$

□

Next I show that conversion is preserved up to equivalence.

Lemma 4.3.5 (Preservation of Conversion). *If* $\Gamma \vdash e \rhd^* e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$

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

Case: Rule RED-REFL, trivial.

Case: Rule RED-TRANS, by Lemma 4.3.4 and the induction hypothesis.

Case: Rule RED-CONG-LET

We have that $\Gamma \vdash \mathsf{let}\; x = e_1 \;\mathsf{in}\; e \rhd^* \mathsf{let}\; x = e_1 \;\mathsf{in}\; e'$ and $\Gamma \vdash e \rhd^* e'$.

We must show that $[\![\Gamma]\!] \vdash [\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; e]\!] \equiv [\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; e']\!]$

$$\begin{aligned}
& [\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; e]\!] \\
= \;& [\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; y[e/y]]\!] && (35)\\
\equiv \;& \mathcal{ST}([\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; y]\!][[\![e]\!]/\mathbf{y}]) \\
& \text{by Lemma 4.3.3 (Substitution)} && (36)\\
\equiv \;& \mathcal{ST}([\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; y]\!][\,[\![e']\!]\,/\mathbf{y}]) \\
& \text{by the induction hypothesis applied to } e \rhd^* e' && (37)\\
\equiv \;& [\![\mathsf{let}\; x = e_1 \;\mathsf{in}\; y[e'/y]]\!]
\end{aligned}$$

$$\begin{aligned} & && \text{by Lemma 4.3.3} && (38) \\ = \;& [\![\mathsf{let}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}']\!] && && (39) \end{aligned}$$

□

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

Lemma 4.3.6 (Preservation of Equivalence). *If* $\Gamma \vdash \mathsf{e} \equiv \mathsf{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \equiv [\![\mathsf{e}']\!]$

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

Case: Rule $\equiv$ Follows by Lemma 4.3.5.

Case: Rule $\equiv$-η_1

By Lemma 4.3.5, we know $[\![\mathsf{e}]\!] \equiv [\![\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1]\!]$. By transitivity, it suffices to show $[\![\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1]\!] \equiv [\![\mathsf{e}']\!]$.

By Rule $\equiv$-η_1, since $[\![\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1]\!] = \boldsymbol{\lambda}\,\mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{e}_1]\!]$, it suffices to show that $[\![\mathsf{e}_1]\!] \equiv [\![\mathsf{e}']\!]\;\mathsf{x}_2$

$$\begin{aligned} & [\![\mathsf{e}_1]\!] && && \\ \equiv\; & [\![\mathsf{e}'\;\mathsf{x}_2]\!] && \text{by the induction hypothesis} && (40) \\ =\; & [\![\mathsf{e}']\!]\;\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,\mathbf{x}_1\;\mathbf{x}_2 && && (41) \\ =\; & (\mathbf{let}\,\mathbf{x}_1 = [\cdot]\,\mathbf{in}\,\mathbf{x}_1\;\mathbf{x}_2)\langle\!\langle\, [\![\mathsf{e}']\!]\,\rangle\!\rangle && \text{by Lemma 4.3.2} && (42) \\ \equiv\; & \mathsf{let}\,\mathsf{x}_1 = [\![\mathsf{e}']\!]\;\mathsf{in}\,\mathsf{x}_1\;\mathsf{x}_2 && \text{by Lemma 4.2.1} && (43) \\ \rhd_\zeta\; & [\![\mathsf{e}']\!]\;\mathsf{x}_2 && && (44) \end{aligned}$$

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

Since I implement cumulative universes through subtyping, we must also show subtyping is preserved. The proof is completely uninteresting, except insofar as it is simple, while it seems to be impossible for CPS translation (Bowman et al., 2018). I discuss this further in Section 4.4.

Lemma 4.3.7 (Preservation of Subtyping). *If* $\Gamma \vdash \mathsf{e} \preceq \mathsf{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \preceq [\![\mathsf{e}']\!]$

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

Case: Rule $\preceq$-$\equiv$. Follows by Lemma 4.3.6.

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

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

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

Case: Rule $\preceq$-Pi.

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

By definition of the translation, we must show $[\![\Gamma]\!] \vdash \mathbf{\Pi}\, \mathbf{x_1} : [\![A_1]\!].\, [\![B_1]\!] \preceq \mathbf{\Pi}\, \mathbf{x_2} : [\![A_2]\!].\, [\![B_2]\!]$.

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

It suffices to show that

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

b) $[\![\Gamma]\!], \mathbf{x_1} : [\![A_2]\!] \vdash [\![B_1]\!] \preceq [\![B_2]\!][\mathbf{x_1}/\mathbf{x_2}]$, which follows by the induction hypothesis.

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

$\square$

I now prove type preservation, with a suitably strengthened induction hypothesis. I prove that, given a well-typed source term e of type A, and a continuation $\mathbf{K}$ that expects the definitions $\texttt{defs}([\![e]\!])$, expects the term $\texttt{hole}([\![e]\!])$, and has result type $\mathbf{B}$, the translation $[\![e]\!]\, \mathbf{K}$ is well typed.

The structure of the lemma and its proof are a little surprising. Intuitively, we would expect to show something like "if $e : A$ then $[\![e]\!] : [\![A]\!]$". I will ultimately prove this, Theorem 4.3.10 (Type Preservation), but we need a stronger lemma first. Since the translation is pushing computation inside-out (since continuations compose inside-out), the type-preservation lemma and proof are essentially inside-out. Instead of the expected statement, we must show that if we have a continuation $\mathbf{K}$ that expects $[\![e]\!] : [\![A]\!]$, then we get a term $[\![e]\!]\, \mathbf{K}$ of some arbitrary type $\mathbf{B}$. Of course, in order to show that, we will have to show that $[\![e]\!] : [\![A]\!]$ and then appeal to Lemma 4.2.4 (Cut). Furthermore, each appeal to the inductive hypothesis will have to establish that we can in fact create well-typed continuations from the assumption that $[\![e]\!] : [\![A]\!]$.

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

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

Lemma 4.3.8. *If for all $\Gamma \vdash e : A$ and $[\![\Gamma]\!], \texttt{defs}([\![e]\!]) \vdash \mathbf{K} : (\texttt{hole}([\![e]\!]) : [\![A]\!]) \Rightarrow \mathbf{B}$ we know that $[\![\Gamma]\!] \vdash [\![e]\!]\, \mathbf{K} : \mathbf{B}$, then $[\![\Gamma]\!], \texttt{defs}([\![e]\!]) \vdash \texttt{hole}([\![e]\!]) : [\![A]\!]$ (and, incidentally, $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$)*

Proof. Note that by Theorem 4.3.1 (Normal Form) and the definitions of $\texttt{defs}([\![e]\!])$ and $\texttt{hole}([\![e]\!])$, $[\![\Gamma]\!], \texttt{defs}([\![e]\!]) \vdash \texttt{hole}([\![e]\!]) : [\![A]\!]$ is a sub-derivation of $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$, so it suffices to show that $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$. By the premise $[\![\Gamma]\!] \vdash [\![e]\!]\,\mathbf{K} : \mathbf{B}$, it suffices to show that $[\cdot] : (\_ : [\![A]\!]) \Rightarrow [\![A]\!]$, which is true by Rule K-Empty. □

Lemma 4.3.9 (Type and Well-formedness Preservation)**.**

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

Proof. The proof is by induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash e : A$. The key cases are the typing rules that use dependency, that is, Rule Snd, Rule App, and Rule Let. I give these cases, although they are essentially similar, and a couple of other representative cases, which are uninteresting. I strongly recommend that the reader read the proof case for Rule Snd, in which I take care to spell out the interesting aspects of the proof.

Case: Rule Prop

We must show that $[\![\Gamma]\!] \vdash [\![\mathsf{Prop}]\!]\,\mathbf{K} : \mathbf{B}$.

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

Note that $\texttt{defs}([\![\mathsf{Prop}]\!]) = \cdot$; this property holds for all values.

By Lemma 4.2.4 (Cut), it suffices to show that

a) $\texttt{hole}([\![\mathsf{Prop}]\!]) = \mathbf{Prop}$, which is true by definition of the translation, and

b) $\mathbf{Prop} : [\![\mathsf{Type}_1]\!]$, which is true by Rule Prop, since $[\![\mathsf{Type}_1]\!] = \mathbf{Type}_1$.

Case: Rule Lam

We must show that $[\![\Gamma]\!] \vdash [\![\lambda\, x : A'.\, e']\!]\,\mathbf{K} : \mathbf{B}$.

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

Recall that $\texttt{defs}([\![\lambda\, x : A'.\, e']\!]) = \cdot$, since values export no definitions.

By Lemma 4.2.4, it suffices to show that $[\![\Gamma]\!] \vdash \boldsymbol{\lambda}\,\mathbf{x} : [\![A']\!].\, [\![e']\!] : [\![\Pi\, x : A'.\, B']\!]$.

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

By Rule Lam, it suffices to show

$[\![\Gamma]\!], \mathbf{x} : [\![A']\!] \vdash [\![e']\!] : [\![B']\!]$.

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

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

Case: Rule Snd

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

That is, by definition of the translation, we must show,

$[\![\Gamma]\!] \vdash [\![e]\!]\ \mathbf{let}\,\mathbf{x'} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\,\mathbf{x'}] : \mathbf{B}$.

Let $\mathbf{K'} = \mathbf{let}\,\mathbf{x'} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\,\mathbf{x'}]$.

Note that we know nothing further about the structure of the term we're trying to type check, $[\![e]\!]\ \mathbf{K'}$. Therefore, we cannot appeal to any typing rules directly. This happens because e is a computation, and the translation of computations composes continuations, which occurs "inside-out". Instead, my proof proceeds "inside out": we build up typing invariants in a well-typed continuation $\mathbf{K'}$ (that is, I build up definitions in an accumulator) and then appeal to the induction hypothesis for e with $\mathbf{K'}$. Intuitively, some later case of the proof that knows more about the structure of e will be able to use this well-typed continuation to proceed.

So, by the induction hypothesis, applied to $\Gamma \vdash e : \Sigma\, x : A'.\, B'$ with $\mathbf{K'}$, to complete this proof case, it suffices to show that: $[\![\Gamma]\!], \texttt{defs}([\![e]\!]) \vdash \mathbf{let}\,\mathbf{x'} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\,\mathbf{x'}] : (\texttt{hole}([\![e]\!]) : [\![\Sigma\, x : A'.\, B']\!]) \Rightarrow \mathbf{B}$

By Rule K-Bind, it suffices to show that

a) $[\![\Gamma]\!], \texttt{defs}([\![e]\!]) \vdash \texttt{hole}([\![e]\!]) : [\![\Sigma\, x : A'.\, B']\!]$, which follows by Lemma 4.3.8 applied to the induction hypothesis for $\Gamma \vdash e : \Sigma\, x : A'.\, B'$

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

To complete this case of the proof, it suffices to show Item (b).

Note $\texttt{defs}([\![\mathsf{snd}\ e]\!]) = (\texttt{defs}([\![e]\!]), \mathbf{x'} = \texttt{hole}([\![e]\!]))$ and $\texttt{hole}([\![\mathsf{snd}\ e]\!]) = \mathbf{snd}\,\mathbf{x'}$.

So, by Lemma 4.2.4 (Cut), given the type of $\mathbf{K}$, it suffices to show that

$[\![\Gamma]\!], \texttt{defs}([\![e]\!]), \mathbf{x'} = \texttt{hole}([\![e]\!]) \vdash \mathbf{snd}\,\mathbf{x'} : [\![B'[\mathsf{fst}\ e/x]]\!]$.

By Lemma 4.3.3, $[\![B'[\mathsf{fst}\ e/\mathbf{x}]]\!] \equiv [\![B']\!][[\![\mathsf{fst}\ e]\!]/\mathbf{x}]$.

By Rule Conv, it suffices to show that $[\![\Gamma]\!], \texttt{defs}([\![e]\!]), \mathbf{x'} = \texttt{hole}([\![e]\!]) \vdash \mathbf{snd}\,\mathbf{x'} : [\![B']\!][[\![\mathsf{fst}\ e]\!]/\mathbf{x}]$.

Note that we cannot show this by the typing rule Rule Snd, since the substitution $[\![B']\!][[\![\mathsf{fst}\ e]\!]/\mathbf{x}]$ copies an apparently arbitrary expression $[\![\mathsf{fst}\ e]\!]$ into the type, instead of the expected sub-expression $\mathbf{fst}\,\mathbf{x'}$. That is, by the typing rules, all we can show is that $\mathbf{snd}\,\mathbf{x'} : [\![B']\!][\mathbf{fst}\,\mathbf{x'}/\mathbf{x}]$, but we must show $\mathbf{snd}\,\mathbf{x'} : [\![B']\!][[\![\mathsf{fst}\ e]\!]/\mathbf{x}]$. The translation has disrupted the dependency on e, changing the type that depended on the specific value e into a type that depends on an apparently arbitrary value $\mathbf{x'}$. This is the problem discussed in Section 4.1. It is also where the definitions we have accumulated in our continuation typing save us. We can show that $[\![\mathsf{fst}\ e]\!] \equiv \mathbf{fst}\,\mathbf{x'}$, under the definitions we have accumulated from continuation typing. This follows by Lemma 4.2.5.

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

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

Case: Rule App

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

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

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

By the inductive hypothesis applied to $\Gamma \vdash e_1 : B'[e_1/x]$, it suffices to show that

$$[\![\Gamma]\!], \texttt{defs}([\![e_1]\!]) \vdash \mathbf{let\, x_1} = [\cdot]\ \mathbf{in}$$
$$[\![e_2]\!]\, \mathbf{let\, x_2} = [\cdot]\, \mathbf{in\, K}[\mathbf{x_1\ x_2}] : (\texttt{hole}([\![e_1]\!]) : [\![\Pi\, x : A'.\, B']\!]) \Rightarrow \mathbf{B}$$

To show this, by Rule K-Bind, it suffices to show

a) $[\![\Gamma]\!], \texttt{defs}([\![e_1]\!]) \vdash \texttt{hole}([\![e_1]\!]) : [\![\Pi\, x : A'.\, B']\!]$, which follows by Lemma 4.3.8 applied to the induction hypothesis for $\Gamma \vdash e_1 : \Pi\, x : A'.\, B'$

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

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

By Rule K-Bind, it suffices to show

$[\![\Gamma]\!], \texttt{defs}([\![e_1]\!]), \mathbf{x_1} = \texttt{hole}([\![e_1]\!]), \texttt{defs}([\![e_2]\!]), \mathbf{x_2} = \texttt{hole}([\![e_2]\!]) \vdash \mathbf{K}[\mathbf{x_1\ x_2}] : \mathbf{B}$.

Let $\mathbf{\Gamma'} = [\![\Gamma]\!], \texttt{defs}([\![e_1]\!]), \mathbf{x_1} = \texttt{hole}([\![e_1]\!]), \texttt{defs}([\![e_2]\!]), \mathbf{x_2} = \texttt{hole}([\![e_2]\!])$, for brevity.

By Lemma 4.2.4 (Cut), we must show $\mathbf{\Gamma'} \vdash \mathbf{x_1\ x_2} : [\![B'[e_2/x]]\!]$.

By Lemma 4.3.3 and Rule Conv, it suffices to show $\mathbf{\Gamma'} \vdash \mathbf{x_1\ x_2} : [\![B']\!][[\![e_2]\!]/x]$.

As in the proof case for Rule Snd, we cannot proceed directly by Rule App, since we see a disrupted dependency. This dependent application whose type depends on the argument being the specific value $[\![e_2]\!]$ now finds the argument $\mathbf{x_2}$. (This issue comes up in type-preservation for call-by-value CPS (Chapter 6).) But, again, we know by Lemma 4.2.5 that under these exported definitions, $[\![e_2]\!] \equiv \mathbf{x_2}$. So by Rule Conv, it suffices to show $\mathbf{\Gamma'} \vdash \mathbf{x_1\ x_2} : [\![B']\!][\mathbf{x_2}/x]$. By Rule App it suffices to show

a) $\mathbf{\Gamma'} \vdash \mathbf{x_1} : \mathbf{\Pi\, x}{:}[\![A']\!].\, [\![B']\!]$, which follows by Lemma 4.3.8 applied to the induction hypothesis for $\Gamma \vdash e_1 : \Pi\, x : A'.\, B'$

b) $\mathbf{\Gamma'} \vdash \mathbf{x_2} : [\![A']\!]$, which follows by Lemma 4.3.8 applied to the induction hypothesis for $\Gamma \vdash e_2 : A'$.

Case: Rule Let

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

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

By the induction hypothesis applied to $\Gamma \vdash e_1 : A$, it suffices to show that

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

By Rule K-Bind, it suffices to show

a) $[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]) \vdash \mathtt{hole}([\![e_1]\!]) : [\![[\![A]\!]]\!]$, which follows by Lemma 4.3.8 applied to the induction hypothesis for $\Gamma \vdash e_1 : A$,

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

Item (b) follows from the induction hypothesis applied to $\Gamma, x = e_1 \vdash e_2 : B'$ with $\mathbf{K}$ (the *same* well-typed $\mathbf{K}$ that we have from our current premise), if we can show that $\mathbf{K}$ is well typed as follows:

$[\![\Gamma]\!], \mathtt{defs}([\![e_1]\!]), \mathbf{x_1} = \mathtt{hole}([\![e_1]\!]), \mathtt{defs}([\![e_2]\!]) \vdash \mathbf{K} : (\mathtt{hole}([\![e_2]\!]) : [\![B'[e_1/x_1]]\!]) \Rightarrow \mathbf{B}$

Currently, we know by our premises that

$[\![\Gamma]\!], \mathtt{defs}([\![\mathsf{let}\, x_1 = e_1\, \mathsf{in}\, e_2]\!]) \vdash \mathbf{K} : (\mathtt{hole}([\![\mathsf{let}\, x_1 = e_1\, \mathsf{in}\, e_2]\!]) : [\![B'[e_1/x_1]]\!]) \Rightarrow \mathbf{B}$

So it suffices to show that

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

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

both of which are straightforward by definition.

□

Theorem 4.3.10 (Type Preservation). *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$

Proof. By Lemma 4.3.9, it suffices to show that $[\![\Gamma]\!] \vdash [\cdot] : (\_ : [\![A]\!]) \Rightarrow [\![A]\!]$, which is trivial. □

4.3.2 Compiler Correctness

I prove correctness of separate compilation with respect to the ANF machine semantics. I use the same definitions of linking and observations as in Chapter 2 for ECC^D. It is simple to lift the ANF translation to closing substitutions.

Correctness of separate compilation is non-standard compared to Chapter 3, in that I define ANF semantics separate from conversion. This theorem states that we can either link then run a program in the source language semantics, *i.e.*, using the conversion relation, or separately compile the term and its closing substitution then run in the ANF machine semantics. Either way, we get related observations.

Theorem 4.3.11 (Separate Compilation Correctness). *If* $\Gamma \vdash e$ *and* $\Gamma \vdash \gamma$, *then* $\mathsf{eval}(\gamma(e)) \approx \mathbf{eval}([\![\gamma]\!]([\![e]\!]))$.

Proof. Because $\equiv$ corresponds to $\approx$ on observations, it suffices to show that the following diagram commutes, which it does because $\mathsf{eval}(\gamma(e)) \equiv \gamma(e)$ (by definiton), $\gamma(e) \equiv [\![\gamma(e)]\!]$

(by Lemma 4.3.6), $[\![\gamma]\!]\,([\![e]\!]) \equiv [\![\gamma(e)]\!]$ (by Lemma 4.3.3), and $\mathbf{eval}([\![\gamma]\!]\,([\![e]\!])) \equiv [\![\gamma]\!]\,([\![e]\!])$ (by Theorem 4.2.3 (Evaluation soundness)).

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

□

Corollary 4.3.12 (Whole-Program Correctness). *If* $\cdot \vdash e$ *then* $\mathsf{eval}(e) \approx [\![\mathbf{eval}(e)]\!]$.

4.4 Related and Future Work

4.4.1 Comparison to CPS

ANF is usually seen in opposition to CPS, so I briefly discuss similarities and differences between our type-preserving ANF and prior work on-type preserving CPS. 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—and summarizes the arguments for and against.

Most recent work on CPS translation of dependently typed language has focused on expressing control effects. Pédrot (2017) uses a non-standard CPS translation to internalize classical reasoning in the Calculus of Inductive Constructions (CIC). Cong and Asai (2018a,b) develop CPS translations to express delimited control effects, via `shift` and `reset`, in a dependently typed language. Miquey (2017) uses a CPS translation to model a dependent sequent calculus. When expressing control effects in dependently typed languages, it is *necessary* to prevent certain dependencies from being expressed to maintain consistency (Barthe and Uustalu, 2002; Herbelin, 2005), therefore these translations do not try to recover dependencies in the way we discuss in Section 4.1.

My own CPS translation, Chapter 6 (published before writing this chapter as Bowman et al. (2018)) does avoid control effects and seeks to develop a type preserving translation. The new typing rule I add is similar to my Rule K-Bind in ECC^A, and is used for the same purpose: to recover disrupted dependencies. Unfortunately, that encoding does not scale to higher universes, and relies on interpreting all functions as parametric (discussed further in Chapter 8). By contrast, this ANF translation works with higher universes and, since ECC^A is a subset of ECC^D, the ANF translation is orthogonal to parametricity.

4.4.2 Branching and Join Points

The ANF translation presented so far does not support all of ECC^D; in particular, it omits the dependent conditional. This is primarily for simplicity, as many of the problems with ANF are orthogonal to dependent conditional. However, dependent conditionals do introduce non-trivial challenges for ANF translation. In this section, I give the translation and argue that it is type preserving.

It is well-known that ANF in the presence of branching constructs, such as `if`, can cause considerable code duplication for branches. For instance, supposing we have an (for the moment, non-dependent) `if`, the naïve ANF translation is the following.

$$[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]\ \mathbf{K} = [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ ([\![\mathsf{e}_1]\!]\ \mathbf{K})\ \mathbf{else}\ ([\![\mathsf{e}_2]\!]\ \mathbf{K})$$

Notice that the current continuation $\mathbf{K}$ is duplicated in the branches.

The well-known solution is to add a *join point*—essentially, a continuation that is used only for avoiding code duplication in branch constructs. Using join points, we would translate if as follows.

$$\begin{aligned}[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]\ \mathbf{K} = [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{let}\ \mathbf{j} = \boldsymbol{\lambda}\, \mathbf{x}' : \mathbf{A}.\ \mathbf{K}[\mathbf{x}']\ \mathbf{in}& \\ \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \ \mathbf{j}\ \mathbf{x}_1& \\ \mathbf{else}\ \ [\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \ \mathbf{j}\ \mathbf{x}_2&\end{aligned}$$

Instead of duplicating $\mathbf{K}$, we create a join point $\mathbf{j}$ which is called in the branches.

Extending the translation to support join points requires (for decidable type checking) that the translation generate a type annotation $\mathbf{A}$ for the join point, where $\mathbf{A}$ is the translation of the type of the $\mathbf{if}$ statement. It is easy to extend the translation to be defined on typing derivations; in fact, the rest of the translation defined in this dissertation are defined on typing derivations, and the proof architecture described in Chapter 3 is designed to support this. The only disadvantage is that structuring the translation this way disallows typed equivalence, unless the problem discussed in Chapter 3 can be solved.

The real problem arises when we have *dependent conditionals*, in which the result type of the branches can depend on the scrutinee of the conditional. Recall that typing rule for dependent `if`, reproduced below.

$$\frac{\Gamma, \mathsf{y} : \mathsf{bool} \vdash \mathsf{B} : \mathsf{U} \qquad \Gamma \vdash \mathsf{e} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{B}[\mathsf{true}/\mathsf{y}] \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{false}/\mathsf{y}]}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}[\mathsf{e}/\mathsf{y}]}$$

We can describe the problem clearly using continuation typing. The ANF translation of this term is with respect to continuation $\mathbf{K} : (\mathtt{hole}([\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]) : [\![\mathsf{B}[\mathsf{e}/\mathsf{y}]]\!]) \Rightarrow \mathbf{B}'$. However, the ANF translation will use $\mathbf{K}$ in an ill-typed way, producing in one branch $[\![\mathsf{e}_1]\!]\ \mathbf{K}$, and in the other branch $[\![\mathsf{e}_2]\!]\ \mathbf{K}$. For the first branch, for instance, we must show that now $\mathbf{K} : (\mathtt{hole}([\![\mathsf{e}_1]\!]) : [\![\mathsf{B}[\mathsf{true}/\mathsf{y}]]\!]) \Rightarrow \mathbf{B}'$. The definitions, $\mathtt{defs}([\![\mathsf{e}]\!])$, introduced by the translation of `if` are not sufficient to show that this type is equivalent to the type of $\mathbf{K}$ expected for the translation of `if`. I discuss an essentially similar problem for CPS with dependent conditionals in Chapter 6.

We could resolve this, it seems, if we could assume that while type checking the first branch that $\mathsf{e} = \mathsf{true}$, and similarly for the second branch that $\mathsf{e} = \mathsf{false}$. But one of these is not true; we cannot have that both $\mathsf{e} = \mathsf{true}$ and $\mathsf{e} = \mathsf{false}$. Since in ECC^A we require reduction under both branches during type checking, reduction under this inconsistent assumption could cause divergence.

To enable us to make inconsistent assumptions like the above but ensure strong normalization, my idea is to use explicit equality proofs and explicit coercions. The coercions would block reduction until they have a proof of equality, and since we will never end up with a proof that $\mathsf{true} = \mathsf{false}$, the inconsistent assumption can never be used in reduction. But the coercion would allow the ANF translation to type check.

With this idea, the type rule for if would be the following.

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \qquad \Gamma, \mathsf{p} : \mathsf{e} = \mathsf{true} \vdash \mathsf{e}_1 : \mathsf{B}[\mathsf{true}/\mathsf{y}] \qquad \begin{array}{c}\Gamma, \mathsf{y} : \mathsf{bool} \vdash \mathsf{B} : \mathsf{U} \\ \end{array} \qquad \Gamma, \mathsf{p} : \mathsf{e} = \mathsf{false} \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{false}/\mathsf{y}]}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}[\mathsf{e}/\mathsf{y}]}$$

Then, the ANF naïve translation for dependent conditionals would be the following; I give the join-point translation shortly.

$$\begin{aligned}
[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]\ \mathbf{K} = [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}& \\
\mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{subst}_{\,\mathbf{p}}\ \mathbf{x}_1]& \\
\mathbf{else}\ \ [\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{subst}_{\,\mathbf{p}}\ \mathbf{x}_2]&
\end{aligned}$$

The term $\mathbf{subst}_{\,\mathbf{p}}\ \mathbf{e}$ is an elimination for the identity type (derivable from only axiom J, a standard axiom admitted in many dependent type theories), with the following standard semantics. Recall that the result type of $\mathbf{K}$ cannot depend on the term in its hole, so we do not need a similar conversion for the result of $\mathbf{K}$.

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

$$\mathbf{subst}_{\,\mathbf{refl}\ \mathbf{e}}\ \mathbf{e} \triangleright \mathbf{e}$$

The translation with join points is below, and requires no further additions.

$$\begin{aligned}
[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2]\!]\ \mathbf{K} = [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{let}\ \mathbf{j} = \lambda\, \mathbf{x}' : \mathbf{A}.\, \mathbf{K}[\mathbf{x}']\ \mathbf{in}& \\
\mathbf{if}\ \mathbf{x}\ \mathbf{then}\ [\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ \mathbf{j}\ \mathbf{subst}_{\,\mathbf{p}}\ \mathbf{x}_1& \\
\mathbf{else}\ \ [\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{j}\ \mathbf{subst}_{\,\mathbf{p}}\ \mathbf{x}_2&
\end{aligned}$$

The type annotation $\mathbf{A}$ would be the translation of the type source if expression. This is why, formally, the translation must be defined over typing derivations instead of syntax.

I conjecture that this translation would be type preserving, but a full investigation is left as future work.

Cong and Asai (2018a) use a similar approach to extend the CPS translation presented in Chapter 6 to inductive types with dependent case analysis.

4.4.3 Dependent Pattern Matching and Commutative Cuts

The above problem with dependent conditionals is exactly the problem of commutative cuts for case analysis (Boutillier, 2012; Herbelin, 2009). Formally, the problem of commutative cuts can be phrased: Is the following transformation type preserving?

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

ANF necessarily performs this transformation, as shown in the previous section.

Ignoring ANF for a moment, in general this *is not* type preserving, and stack typing shows why. Suppose that we have a non-ANF stack K with the following typing derivation.

$$\frac{\Gamma, \mathsf{y}' : \mathsf{bool} \vdash \mathsf{B}' : \mathsf{U}' \qquad \Gamma, \mathsf{y} : \mathsf{B}'[\mathsf{e}/\mathsf{y}'] \vdash \mathsf{B} : \mathsf{U} \qquad \Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}'[\mathsf{e}/\mathsf{y}']}{\Gamma \vdash \mathsf{K} : (\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}'[\mathsf{e}/\mathsf{y}']) \Rightarrow \mathsf{B}[(\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)/\mathsf{y}]}$$

Note that the result type of this stack, B', *may* depend on the term in the hole, since this is not in ANF. The problem is that after the commutative cut, we must show that $\mathsf{K}[\mathsf{e}_1]$ and $\mathsf{K}[\mathsf{e}_2]$ are well-typed in the branches, which is not true in general since K expects $\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$.

If we add the equalities $\mathsf{e} = \mathsf{true}$ and $\mathsf{e} = \mathsf{false}$ while type checking the branches, as proposed in the previous section, then it appears that we can make the terms $\mathsf{K}[\mathsf{e}_1]$ and $\mathsf{K}[\mathsf{e}_2]$ well-typed, although it requires a generalization of Lemma 4.2.4. However, we still cannot show the commutative cut is type preserving, since the result type B also depends on the term in the hole. And now, the two branches of the if have different types: $\mathsf{K}[\mathsf{e}_1] : \mathsf{B}[\mathsf{e}_1/\mathsf{y}]$, while $\mathsf{K}[\mathsf{e}_2] : \mathsf{B}[\mathsf{e}_2/\mathsf{y}]$. Since the branches of an if must have the same type (up to equivalence), it appears that we must show $\mathsf{e}_1 \equiv \mathsf{e}_2$.

In fact, what we need to show is essentially another, smaller, commutative cut. Viewing B as a type-level stack, we must show $\mathsf{B}[\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2] \equiv \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{B}[\mathsf{e}_1]\ \mathsf{else}\ \mathsf{B}[\mathsf{e}_2]$. This is smaller in the sense that the type of this type cannot contain a commutative cut. For booleans, we could pursue this by adding yet another appeal to J, but this approach does not scale to indexed inductive types.

But, there is a solution: we can make the commutative cut type preserving, even for general inductive types. Boutillier (2012) give an extension to CIC that allows for typing commutative cuts, in particular, by relaxing the termination checker. Explaining the solution is out of scope for this work. I only want to point out two things. First, the solution adds an environment of *equalities* to the termination checker, just as I propose for stack typing and for typing if above. Second, the solution requires *axiom* K, a very strong requirement which is inconsistent with certain extensions to type theory, such as univalence. This raises the question: does ANF in general require axiom K?

I conjecture the answer is no, and the reason is the interesting property we observed in Section 4.2: in ANF, the result type cannot depend on the term in the hole. This additional structure seems to avoid some problems of commutative cuts, and I hope that it will be enough to scale ANF to indexed inductive types without additional requirements on the type theory. I have one additional reason to be hopeful: even if we need a stronger axiom than J, work on dependent pattern matching suggests that univalence may replace axiom K for our purposes.

Recent work on dependent pattern matching creates typing rules similar to what we suggest above to yield additional equalities during a pattern match (Barras et al., 2008; Cockx et al., 2016). There is another unfortunate similarity: some work on dependent pattern matching requires axiom K. In particular, Barras et al. (2008) give a new eliminator for CIC which adds additional equalities while checking branches of an elimination, and show that this new typing rule is equivalent to axiom K. Cockx et al. (2016) discuss a proof-relevant view of unification, in the context of Agda's dependent pattern matching. They note that the heterogeneous equalities usually required by dependent pattern matching require axiom K to be useful. They also take a different approach, and build on an idea from homotopy type theory that one equality can "layer over" another, to get a proof relevant unification algorithm that does not rely on K, and yet yields the additional equalities for dependent pattern matching.

4.4.4 Dependent Call-By-Push-Value and Monadic Form

Call-by-push-value ($CBPV$) is similar to the ANF target language, and to CPS target languages. In essence, CBPV is a λ calculus in monadic normal form suitable for reasoning about CBV or CBN, due to explicit sequencing of computations. It has values, computations, and stacks, similar to ANF, and has compositional typing rules (which inspired much of my own presentation). The particular structure of CBPV is beneficial when 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 Section 4.1. The reason is essentially that monadic normal form is more compositional than CPS or ANF, so dependency is not disrupted in the same way.

Recall from Section 4.2 that our definition of composition was entirely motivated by the need to compose configurations and stacks. 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 stack inside the body of a `let`. The monadic translation of $\mathsf{snd}\,\mathsf{e}$ from Section 4.1, which is problematic in CPS and ANF, is given below and is easily type preserving.

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

Note that since `let` can bind the "configuration" $[\![\mathsf{e}]\!]$, the typing rule LET and the compositionality lemma suffice to show type preservation, without any reasoning about definitions. In fact, we don't even need *definitions* for monadic form; we only need a dependent result type for **let**. The dependent typing rule for **let** without definition 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. A similar observation was made independently in my own work with Nick Rioux, Youyou Cong, and Amal Ahmed (2018, presented in Chapter 6): 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).

If we restrict the language so that types can only depend on *values*, then the extension with dependent `let` is not necessary. This restriction seems sensible in the context of modeling effects. Ahman (2017) in *eMLTT*, a co-discovered variant of dependent CBPV, avoids dependent let altogether, but comes up with many useful models of dependent types with effects. Ahman (2017), however, does not give a translation from type theory into eMLTT, and it seems likely that an extension with dependent let would be required to do so. (This would be necessary to build on eMLTT as a compiler IL.) However, as Vákár (2017) points out, it is not clear what it *means* to have a type depend on an effectful computation, and trying to do so makes it impossible to model effects in dCBPV the way one would hope.

In eMLTT, stacks cannot have a result type that depends on the value it is composed with, just as in our K-BIND rule. However, the dCBPV of Vákár (2017) *does* allow the result type of stacks to depend on values, but only on values. It is unclear what trade-offs each approach presents.

5 ABSTRACT CLOSURE CONVERSION

In this chapter, I develop the second of the two front-end type-preserving translations, a so-called abstract closure conversion translation. *Abstract closure conversion* produces closure converted code in which closures are primitives, rather than encoded using a well-known datatype (Minamide et al., 1996). The goal of *closure conversion* is to close all computation abstractions with respect to free variables so the definition of a computation can be separate from its use. In ECC^{D}, the only computation abstraction is λ, so we will be translating all λ expressions into an explicit closure object. Intuitively, the abstract closure object essentially represents closed code partially applied to the local environment in which the code was defined. This is abstract compared to representations of closures as a pair of the code and environment, since a pair supports more operations (projection) than a closure. The abstract closure conversion ensures the only operation defined on closures is application.

I begin by describing the key problems with type preserving closure conversion—particularly why the standard parametric closure conversion fails—and the solutions, then develop the closure conversion IL, and finally prove type preservation and compiler correctness. As this pass is meant to follow ANF, I also prove that ANF is preserved.

Typographical Note. *In this chapter, I typeset the source language, ECC^{D}, in a* blue, non-bold, sans-serif font, *and the target language, ECC^{CC}, in a* **bold, red, serif font**.

Digression. *A variant of the translation presented in this chapter was independently discovered by Kovács (2018). That translation differs primarily in its use of universes ala Tarski. The author also spends some time discussing type-passing polymorphism, which I do not discuss.*

5.1 Main Ideas

Closure conversion makes the implicit closures from a functional language explicit to facilitate statically allocating code in memory. The idea is to translate each first-class function into an explicit closure, *i.e.*, a pair of closed code and an *environment* data structure containing the values of the free variables. *Code* refers to functions with no free variables, as in a closure-converted language. The environment is created dynamically, but the closed code can be lifted to the top-level and statically allocated. Consider the following example translation.

$$\begin{aligned}
[\![(\lambda x.y)]\!] &= \langle(\lambda n\, x.\, \mathrm{let}\, y = (\pi_1\, n) \text{ in } y), \langle y \rangle\rangle \\
[\![((\lambda x.y) \text{ true})]\!] &= \text{let } \langle f, n\rangle = \langle(\lambda n\, x.\, \mathrm{let}\, y = (\pi_1\, n) \text{ in } y), \langle y \rangle\rangle \text{ in} \\
& \quad\; f\ n \text{ true}
\end{aligned}$$

I write $[\![e]\!]$ to indicate the translation of an expression e. We translate each function into a pair of code and its environment. The code accepts its free variables in an environment argument, n (since n sounds similar to *env*). In the body of the code, we bind the names of all free variables by projecting from this environment n. To call a closure, we apply the code to its environment and its argument.

This translation is not type preserving since the structure of the environment shows up in the type. For example, the following two functions have the same type in the source language. They are both functions on booleans, and y is a free variable of type bool.

$$\begin{aligned}
\lambda x.y &: (\text{bool} \to \text{bool}) \\
\lambda x.x &: (\text{bool} \to \text{bool})
\end{aligned}$$

But after closure conversion when we encode closures as pairs, the two functions have different types in the target language.

$$\begin{aligned}
[\![(\lambda x.y)]\!] &: ((\text{bool} \times 1) \to \text{bool} \to \text{bool}) \times (\text{bool} \times 1) \\
[\![(\lambda x.x)]\!] &: (1 \to \text{bool} \to \text{bool}) \times 1
\end{aligned}$$

This is a well-known problem with typed closure conversion, so we could try the well-known solution, called *parametric closure conversion* [Minamide et al., 1996, Morrisett and Harper, 1998, Morrisett et al., 1999, Ahmed and Blume, 2008, Perconti and Ahmed, 2014, New et al., 2016], which represents closures as an existential package of a pair of the code and its environment, whose type is hidden. The existential type hides the structure of the environment in the type. (Spoiler alert: it doesn't work for ECC^D.)

$$\begin{aligned}
[\![(\lambda x.y)]\!] &: \exists\alpha.(\alpha \to \text{bool} \to \text{bool}) \times \alpha \\
[\![(\lambda x.x)]\!] &: \exists\alpha.(\alpha \to \text{bool} \to \text{bool}) \times \alpha
\end{aligned}$$

This translation works well for simply typed and polymorphic languages, but when we move to a dependently typed language, we have new challenges. First, the environment must now be ordered since the type of each new variable can depend on all prior variables. Second, types can now refer to variables in the closure's environment. Consider the polymorphic identity function below.

$$\lambda\, \mathsf{A} : \mathsf{Prop}\,.\, \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{x} \quad : \quad \Pi\, \mathsf{A} : \mathsf{Prop}\,.\, \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{A}$$

This function takes a type variable, A, whose type is Prop. It returns a function that accepts an argument x of type A and returns it. There are two closures in this example:

the outer closure has no free variables, and thus will have an empty environment, while the inner closure $\lambda\, x : A.\, x$ has A free, and thus A will appear in its environment.

Below, I present the translation of this example using the existing parametric closure conversion translation. The translation produces two closures, one nested in the other. Note that we translate source variables x to $\mathbf{x}$. In the outer closure, the environment is empty $\langle\rangle$, and the code simply returns the inner closure. The inner closure has the argument $\mathbf{A}$ from the outer code in its environment. Since the inner code takes an argument of type $\mathbf{A}$, we project $\mathbf{A}$ from the environment *in the type annotation* for $\mathbf{x}$. That is, the inner code takes an environment $\mathbf{n_2}$ that contains $\mathbf{A}$, and the type annotation for $\mathbf{x}$ is $\mathbf{x} : \mathbf{fst}\,\mathbf{n_2}$. The type $\mathbf{fst}\,\mathbf{n_2}$ is unusual, but is no problem since dependent types allow terms in types.

$$\begin{aligned}
&\langle\!\langle \boldsymbol{\lambda}\,(\mathbf{n_1} : \mathbf{1}, \mathbf{A} : \mathbf{Prop}).\, \langle\!\langle \boldsymbol{\lambda}\,(\mathbf{n_2} : \mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\,\mathbf{n_2}).\,\mathbf{x}, \langle \mathbf{A}, \langle\rangle\rangle\rangle\!\rangle, \langle\rangle\rangle\!\rangle \quad : \\
&\quad \exists\, \boldsymbol{\alpha}_1 : \mathbf{Prop}\,.\,(\boldsymbol{\Pi}\,(\mathbf{n_1} : \boldsymbol{\alpha}_1, \mathbf{A} : \mathbf{Prop}\,). \\
&\qquad\qquad\qquad \exists\, \boldsymbol{\alpha}_2 : \mathbf{Type}_{\,i}.\,(\boldsymbol{\Pi}\,(\mathbf{n_2} : \boldsymbol{\alpha}_2, \mathbf{x} : \mathbf{fst}\,\mathbf{n_2}).\,\mathbf{fst}\,\mathbf{n_2}) \times \boldsymbol{\alpha}_2) \times \boldsymbol{\alpha}_1
\end{aligned}$$

We see that the inner code on its own is well typed with the closed type $\boldsymbol{\Pi}\,(\mathbf{n_2} : \mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\,\mathbf{n_2}).\,\mathbf{fst}\,\mathbf{n_2}$. That is, the code takes two arguments: the first argument $\mathbf{n_2}$ is the environment, and the second argument $\mathbf{x}$ is a value of type $\mathbf{fst}\,\mathbf{n_2}$. The result type of the code is also $\mathbf{fst}\,\mathbf{n_2}$. As discussed above, we must hide the type of the environment to ensure type preservation. That is, when we build the closure $\langle\!\langle \boldsymbol{\lambda}\,(\mathbf{n_2} : \mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\,\mathbf{n_2}).\,\mathbf{x}, \langle \mathbf{A}, \langle\rangle\rangle\rangle\!\rangle$, we must hide the type of the environment $\langle \mathbf{A}, \langle\rangle\rangle$. We use an existential type to quantify over the type $\boldsymbol{\alpha}_2$ of the environment, and we produce the type $\boldsymbol{\Pi}\,(\mathbf{n_2} : \boldsymbol{\alpha}_2, \mathbf{x} : \mathbf{fst}\,\mathbf{n_2}).\,\mathbf{fst}\,\mathbf{n_2}$ for the code in the inner closure. But this type is trying to take the *first projection* of something of type $\boldsymbol{\alpha}_2$. We can only project from pairs, and something of type $\boldsymbol{\alpha}_2$ isn't a pair! In hiding the type of the environment to recover type preservation, we've broken type preservation for dependent types.

A similar problem also arises when closure converting System F, since System F also features type variables (Minamide et al., 1996; Morrisett et al., 1999). To understand my solution, it is important to understand why the solutions that have historically worked for System F do not scale to dependent types. I briefly present these past results and why they do not scale before moving on to the key idea behind my translation. Essentially, past work using existential types relies on assumptions about computational relevance, parametricity, and impredicativity that do not necessarily hold in full-spectrum dependent type systems.

5.1.1 Why the Well-Known Solution Doesn't Work

Minamide et al. (1996) give a translation that encodes closure types using existential types, a standard type-theoretic feature that they use to make environment hiding explicit in the types. In essence, they encode closures as objects; the environment can be thought of as the private field of an object. Since then, existential types have been the standard way to encode closure types has been all work on typed closure conversion.

However, the use of existential types to encode closures in a dependently typed setting is problematic. First, let us just consider closure conversion for System F. As Minamide et al. (1996) observed, there is a problem when code must be closed with respect to both term and *type* variables. This problem is similar to the one discussed above: when closure environments contain type variables, since those type variables can also appear in the closure's type, the closure's type needs to project from the closure's (hidden) environment which has type α. To fix the problem, Minamide et al. (1996) extend their target language with *translucency* (essentially, a kind of type-level equivalence that we now call singleton types), type-level pairs, and kinds. All of these features can be encoded in ECC^D and most dependent type systems, so we could extend their translation essentially as follows. (In fact, I present the extended translation in Chapter 7.)

$$[\![(\Pi\, x : A.\, B)]\!] \overset{\text{def}}{=} \exists\, \boldsymbol{\alpha} : \mathbf{U}.\, \exists\, \mathbf{n} : \boldsymbol{\alpha}.\, \mathbf{Code}\,(\mathbf{n'} : \boldsymbol{\alpha}, \mathbf{y} : \mathbf{n'} = \mathbf{n}, \mathbf{x} : [\![A]\!]).\, [\![B]\!]$$

In this translation, we existentially quantify over the type of the environment $\boldsymbol{\alpha}$, the *term* representing the environment $\mathbf{n}$, and generate code that requires an environment $\mathbf{n'}$ plus a proof that the code is only ever given the environment $\mathbf{n}$ as the argument $\mathbf{n'}$. The typing rule for an existential package copies the existentially quantified terms into the type. That is, for a closure $\mathbf{pack}\,\langle \mathbf{A'}, \mathbf{v}, \mathbf{e}\rangle$ of type $\exists\, \boldsymbol{\alpha} : \mathbf{U}.\, \exists\, \mathbf{n} : \boldsymbol{\alpha}.\, \mathbf{Code}\,(\mathbf{n'} : \boldsymbol{\alpha}, \mathbf{y} : \mathbf{n'} = \mathbf{n}, \mathbf{x} : [\![A]\!]).\, [\![B]\!]$, the typing rule for $\mathbf{pack}$ requires that we show $\mathbf{e} : \mathbf{Code}\,(\mathbf{n'} : \mathbf{A'}, \mathbf{y} : \mathbf{n'} = \mathbf{v}, \mathbf{x} : [\![A]\!]).\, [\![B]\!]$; notice that the variable $\mathbf{n}$ has been replaced by the term representing the environment $\mathbf{v}$. The equality $\mathbf{n'} = \mathbf{v}$ essentially unifies projections from $\mathbf{n'}$ with projections from $\mathbf{v}$, the list of free variables representing the environment.

The problem with this translation is that it relies on *impredicativity*. That is, if $(\Pi\, x : A.\, B) : \mathsf{Prop}$, then we require that $[\![(\Pi\, x : A.\, B)]\!] : \mathbf{Prop}$. Since the existential type quantifies over a type in an arbitrary universe $\mathbf{U}$ but must be in the base universe $\mathbf{Prop}$, the existential type must be impredicative. Impredicative existential types (weak dependent pairs) are consistent on their own, but impredicativity causes inconsistency when combined with other features, including computational relevance and higher universes. For example, in Coq by default, the base computationally relevant universe `Set` is predicative, so this translation would not work. There is a flag to enable impredicative `Set`, but this can introduce inconsistency with some axioms, such as a combination of the law of excluded middle plus the axiom of choice, or ad-hoc polymorphism (Boulier et al., 2017). Even with impredicative `Set`, there are computationally relevant higher universes in Coq's universe hierarchy, and it would be unsound to allow impredicativity at more than one universe. Furthermore, some dependently typed languages, such as Agda, do not allow impredicativity at all.

A second problem arises in developing an η principle, because the existential type encoding relies on *parametricity* to hide the environment. So, any η principle would need to be justified by a parametric relation on environments. Internalizing parametricity for dependent type theory is an active area of research (Krishnaswami and Dreyer, 2013; Bernardy et al., 2012; Keller and Lasson, 2012; Nuyts et al., 2017) and not all dependent type theories admit parametricity (Boulier et al., 2017).

Later, Morrisett et al. (1999) improved the existential-type translation for System F, avoiding translucency and kinds by relying on *type erasure* before runtime, which meant that their code didn't have to close over type variables. This translation does not apply in a dependently typed setting, since dependent types can contain term variables, not just "type erasable" type variables.

5.1.2 Abstract Closure Conversion

To solve type-preserving closure conversion for ECC^D, I avoid existential types altogether and instead take inspiration from the so-called abstract closure conversion of Minamide et al. (1996). They add new forms to the target language to represent code and closures for a simply typed source language. In this chapter, I scale their design to dependent types.

I extend ECC^D with primitive types for code and closures. I represent code as $\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_1$ of the *code type* $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B}$. These are still dependent types, so $\mathbf{n}$ may appear in both $\mathbf{A}$ and $\mathbf{B}$, and $\mathbf{x}$ may appear in $\mathbf{B}$. Code must be well-typed in an empty environment, *i.e.*, when it is closed. For simplicity, code only takes two arguments.

$$\frac{\cdot, \mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A} \vdash \mathbf{e} : \mathbf{B}}{\boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}\,\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}.\, \mathbf{e} : \mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B}}\ \textsc{Code}$$

I represent closures as $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ of type $\boldsymbol{\Pi}\,\mathbf{x} : \mathbf{A}[\mathbf{e}'/\mathbf{n}].\, \mathbf{B}[\mathbf{e}'/\mathbf{n}]$, where $\mathbf{e}$ is code and $\mathbf{e}'$ is its environment. We *continue* to use $\boldsymbol{\Pi}$ types to describe closures; note that "functions" in ECC^D are implicit closures. The typing rule for closures is the following.

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

We should think of a closure $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ not as a pair, but as a delayed partial application of the code $\mathbf{e}$ to its environment $\mathbf{e}'$. This intuition is formalized in the typing rule since the environment is substituted into the type, just as in dependent-function application in ECC^D.

To understand the translation, let us start with the translation of functions; this is the key translation rule.

$$\begin{aligned} [\![(\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e})]\!] \overset{\text{def}}{=}\ & \langle\!\langle (\boldsymbol{\lambda}\,(\mathbf{n} : \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots).\, \mathbf{1}, \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{A}]\!]). \\ & \quad \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{e}]\!]), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle \\ & \text{where } \mathsf{x}_i : \mathsf{A}_i \ldots \text{ are the free variables of } \mathsf{e} \text{ and } \mathsf{A} \end{aligned}$$

The translation of functions is simple to construct. We know we want to produce a closure containing code and its environment. We know the environment should be constructed from the free variables of the body of the function, namely e, and, due to dependent types, the type annotation A. The type of the environment, $\boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots).\,\mathbf{1}$, is the type of a dependent list describing the free variables, with a final element of the unit type. This encodes the fact that type of each variable in the environment can depend on the value of previous variables. (The syntax $\mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e}$ is syntactic sugar for nested projections from a list implemented with pairs, *i.e.*, for $\mathbf{let}\ \mathbf{x}_1 = \mathbf{fst}\ \mathbf{n}\ \mathbf{in}\ (\mathbf{let}\ \mathbf{x}_2 = \mathbf{fst}\ \mathbf{snd}\ \mathbf{n}\ \mathbf{in}\ (\ldots \mathbf{let}\ \mathbf{x}_n = (\mathbf{fst}\ \mathbf{snd}\ \ldots \mathbf{snd}\ \mathbf{n}\ \mathbf{in}\ \mathbf{e})).)$

The question is what the translation of $\mathsf{\Pi}$ types should look like. Let's return to the earlier example of the polymorphic identity function. If we apply the above translation, we produce the following for the inner closure. We know its type by following the typing rules CLO and CODE above.

$$\begin{aligned}&\langle\!\langle \boldsymbol{\lambda}\,(\mathbf{n}_2 : \mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\ \mathbf{n}_2).\,\mathbf{x}, \langle \mathbf{A}, \langle\rangle\rangle \rangle\!\rangle : \\ &\boldsymbol{\Pi}\,(\mathbf{x} : (\mathbf{fst}\ \mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle\rangle/\mathbf{n}_2]).\,(\mathbf{fst}\ \mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle\rangle/\mathbf{n}_2]\end{aligned}$$

We know that the code $\boldsymbol{\lambda}\,(\mathbf{n}_2 : \mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\ \mathbf{n}_2).\,\mathbf{x}$ has type $\mathbf{Code}\,(\mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst}\ \mathbf{n}_2).\,\mathbf{fst}\ \mathbf{n}_2$. Following CLO, we substitute the environment into this type, so we get the following.

$$\boldsymbol{\Pi}\,(\mathbf{x} : (\mathbf{fst}\ \mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle\rangle/\mathbf{n}_2]).\,(\mathbf{fst}\ \mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle\rangle/\mathbf{n}_2]$$

So how do we translate the function type $\mathsf{\Pi\,x : A.\,A}$ into the closure type $\boldsymbol{\Pi}\,(\mathbf{x} : (\mathbf{fst}\ \mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle\rangle/\mathbf{n}_2]).\,(\mathbf{fst}\ \mathbf{n}_2)[\langle \mathbf{A}, \langle\rangle\rangle/\mathbf{n}_2]$? Note that this type reduces to $\boldsymbol{\Pi}\,\mathbf{x} : \mathbf{A}.\,\mathbf{A}$. So by the rule CONV, we simply need to translate $\mathsf{\Pi\,x : A.\,A}$ to $\boldsymbol{\Pi}\,\mathbf{x} : \mathbf{A}.\,\mathbf{A}$!

The key translation rules are given below.

$$\begin{aligned}[\![(\mathsf{\Pi\,x : A.\,B})]\!] &\overset{\text{def}}{=} \boldsymbol{\Pi}\,\mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!] \\ [\![(\mathsf{\lambda\,x : A.\,e})]\!] &\overset{\text{def}}{=} \langle\!\langle (\boldsymbol{\lambda}\,(\mathbf{n} : \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{A}]\!]). \\ &\qquad \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{e}]\!]), \langle \mathbf{x}_i \ldots\rangle \rangle\!\rangle \\ &\quad \text{where } \mathsf{x}_i : \mathsf{A}_i \ldots \text{ are the free variables of } \mathsf{e} \text{ and } \mathsf{A}\end{aligned}$$

Observe that, when the source is in ANF, this translation maintains ANF. A closure ought to be a value, and therefore its sub-expressions should be values, and the translation guarantees this. The body of the code should be a configuration, which is also true, since for any configuration $\mathbf{M}$, $\mathbf{let}\ \mathbf{x} = \mathbf{N}\ \mathbf{in}\ \mathbf{M}$ is a configuration. I show this in detail in Section 5.3.3.

A final challenge remains in the design of our target language: we need to know when two closures are equivalent. As we just saw, ECC^D partially evaluates terms while type checking. If two closures get evaluated while resolving type equivalence, we may inline a term into the environment for one closure but not the other. When this happens, two closures that were syntactically identical and thus equivalent become inequivalent. I discuss this problem in detail in Section 5.3, but essentially we need to know when

$$
\begin{array}{llcl}
\textit{Universes} & \mathbf{U} & ::= & \mathbf{Prop} \mid \mathbf{Type}_i \\
\textit{Expressions} & \mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \mathbf{U} \mid \mathbf{Code}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B} \mid \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e} \\
& & \mid & \boldsymbol{\Pi}\,\mathbf{x} : \mathbf{A}.\, \mathbf{B} \mid \langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle \mid \mathbf{e}\ \mathbf{e}' \mid \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\, \mathbf{B} \\
& & \mid & \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\, \mathbf{B} \mid \mathbf{fst}\,\mathbf{e} \mid \mathbf{snd}\,\mathbf{e} \mid \mathbf{bool} \mid \mathbf{true} \\
& & \mid & \mathbf{false} \mid \mathbf{if}\,\mathbf{e}\,\mathbf{then}\,\mathbf{e}_1\,\mathbf{else}\,\mathbf{e}_2 \mid \mathbf{let}\,\mathbf{x} = \mathbf{e}\,\mathbf{in}\,\mathbf{e} \mid \mathbf{1} \mid \langle\rangle
\end{array}
$$

Figure 5.1: ECC^{CC} Syntax

two syntactically distinct closures are equivalent. The solution is simple: get rid of the closures and keep inlining things!

$$
\frac{\boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A} \vdash \mathbf{e}_1[\mathbf{e}_1'/\mathbf{n}] \equiv \mathbf{e}_2[\mathbf{e}_2'/\mathbf{n}]}{\boldsymbol{\Gamma} \vdash \langle\!\langle (\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_1), \mathbf{e}_1' \rangle\!\rangle \equiv \langle\!\langle (\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_2), \mathbf{e}_2' \rangle\!\rangle}
$$

Two closures are equivalent when we inline the environment, free variables or not, and run the body of the code. We leave the argument free, too. We run the bodies of the code to normal forms, then compare the normal forms. Recall that equivalence runs terms while *type checking* and does not change the program, so the free variables do no harm.

This equivalence essentially corresponds to an η-principle for closures. From it, we can derive a normal form for closures $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ that says the environment $\mathbf{e}'$ contains only free variables, *i.e.*, $\mathbf{e}' = \langle \mathbf{x}_i \ldots \rangle$.

The above is an intuitive, declarative presentation, but is incomplete without additional rules. I use an algorithmic presentation that is similar to the η-equivalence rules for functions in ECC^D, which I give in Section 5.2.

5.2 Closure-Converted Intermediate Language

The target language ECC^{CC} is based on ECC^D, but first-class functions are replaced by closed code and closures. I add a primitive unit type $\mathbf{1}$ to support encoding environments. In Figure 5.1, I extend the syntax of expressions with a unit expression $\langle\rangle$ and its type $\mathbf{1}$, closed code expressions $\boldsymbol{\lambda}\,\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}.\, \mathbf{e}$ and dependent code types $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B}$, and closure expressions $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ and dependent closure types $\boldsymbol{\Pi}\,\mathbf{x} : \mathbf{A}.\, \mathbf{B}$. The closed code expressions will eventually be separated from closures, lifted to the top-level and heap allocated, as described in Chapter 3. The syntax of application $\mathbf{e}\ \mathbf{e}'$ is unchanged, but it now applies closures instead of functions.

I define additional syntactic sugar for sequences of expressions, to support writing environments whose length is arbitrary. A sequence of expressions $\mathbf{e}_i \ldots$ represents a sequence of length $|i|$ of expressions $\mathbf{e}_{i_0}, \ldots, \mathbf{e}_{i_n}$. I extend the notation to patterns such as $\mathbf{x}_i : \mathbf{A}_i \ldots$, which implies two sequences $\mathbf{x}_{i_0}, \ldots, \mathbf{x}_{i_n}$ and $\mathbf{A}_0, \ldots, \mathbf{A}_{i_n}$ each of length

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

$$\vdots$$

$$\Gamma \vdash \langle\!\langle \lambda\, x' : A', x : A.\, e_1, e' \rangle\!\rangle\ e \quad \triangleright_\beta \quad e_1[e'/x'][e/x]$$

Figure 5.2: ECC^{CC} Reduction (excerpts)

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \triangleright^* \langle\!\langle \lambda\, (x' : A', x : A).\, e_1', e' \rangle\!\rangle \\ \Gamma \vdash e_2 \triangleright^* e_2' \qquad \Gamma, x : A \vdash e_1[e'/x'] \equiv e_2'\ x\end{array}}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-Clo}_1$$

$$\frac{\begin{array}{c}\Gamma \vdash e_2 \triangleright^* \langle\!\langle \lambda\, (x' : A', x : A).\, e_2', e' \rangle\!\rangle \\ \Gamma \vdash e_1 \triangleright^* e_1' \qquad \Gamma, x : A \vdash e_1'\ x \equiv e_2'[e'/x']\end{array}}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-Clo}_2$$

Figure 5.3: ECC^{CC} Equivalence

$|i|$. I define environments as dependent n-tuples, written $\langle e_i \ldots \rangle\ \textbf{as}\ \Sigma\,(x_i : A_i \ldots)$. I encode dependent n-tuples using dependent pairs—$\langle e_0, \langle \ldots, \langle e_i, \langle\rangle\rangle\rangle\rangle$— *i.e.*, as nested dependent pairs followed by a unit expression to represent the empty n-tuple. Similarly, the type of a dependent n-tuple is a nested dependent pair type; $\Sigma\,(x_i : A_i \ldots)$ is syntactic sugar for $\Sigma\, x_0 : A_0. \ldots \Sigma\, x_i : A_i.\, 1$.

As with dependent pairs, I omit the annotation on n-tuples $\langle e_i \ldots \rangle$ when it is obvious from context. I also define pattern matching on n-tuples, written $\textbf{let}\ \langle x_i \ldots \rangle = e'\ \textbf{in}\ e$, to perform the necessary nested projections, *i.e.*, $\textbf{let}\ x_0 = \textbf{fst}\ e'\ \textbf{in}\ \ldots \textbf{let}\ x_i = \textbf{fst}\ \textbf{snd}\ \ldots \textbf{snd}\ e'\ \textbf{in}\ e$, as described in Section 5.1.

In Figure 5.2, I present the reduction rules for closures. The conversion relation $\Gamma \vdash e \triangleright^* e'$, and evaluation function $\textbf{eval}(e)$ are essentially unchanged from ECC^D, and are given in full in Appendix D. Note that β-reduction only applies to closures. Code cannot be applied directly, but must be part of a closure. Closures applied to an argument β-reduce, applying the underlying code to the environment and the argument. All the other reduction rules remain unchanged.

In Figure 5.3, I present equivalence for ECC^{CC}. The key difference is that I replace the η-equivalence rules for functions by η-equivalence for closures. Even when η-equivalence for functions is excluded from the source language, η-equivalence for closures is necessary for proving Lemma 4.3.3 (Substitution).

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \equiv A_2 \\ \Gamma, n_1 : A_1 \vdash A_1' \equiv A_2' \qquad \Gamma, n_1 : A_1, x_1 : A_1' \vdash B_1 \preceq B_2[n_1/n_2][x_1/x_2]\end{array}}{\Gamma \vdash \mathbf{Code}\,(n_1 : A_1, x_1 : A_1').\,B_1 \preceq \mathbf{Code}\,(n_2 : A_2, x_2 : A_2').\,B_2}\ \preceq\text{-Code}$$

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

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

Figure 5.4: ECC^{CC} Subtyping

In Figure 5.4, I define subtyping. As usual, subtyping extends equivalence to include cumulativity. The subtyping rules for code types are essentially the same as for dependent function types.

$$\cdots \qquad \frac{\Gamma, x' : A', x : A \vdash B : \mathbf{Prop}}{\Gamma \vdash \mathbf{Code}\,(x' : A', x : A).\, B : \mathbf{Prop}}\ \text{T-Code-Prop}$$

$$\frac{\Gamma, x' : A', x : A \vdash B : \mathbf{Type}_i}{\Gamma \vdash \mathbf{Code}\,(x : A, x' : A').\, B : \mathbf{Type}_i}\ \text{T-Code-Type}$$

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

$$\frac{\Gamma \vdash e : \mathbf{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash \langle\!\langle e, e' \rangle\!\rangle : \Pi\, x : A[e'/x'].\, B[e'/x']}\ \text{Clo} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{1} : \mathbf{Prop}}\ \text{T-Unit}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \langle\rangle : \mathbf{1}}\ \text{Unit}$$

Figure 5.5: ECC^{CC} Typing (excerpts)

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

$$
\begin{aligned}
[\![\mathbf{1}]\!]^{\circ} &\overset{\text{def}}{=} \Pi\, \alpha : \mathsf{Prop}\,.\, \Pi\, \mathsf{x} : \alpha.\, \alpha \\
[\![\langle\rangle]\!]^{\circ} &\overset{\text{def}}{=} \lambda\, \alpha : \mathsf{Prop}\,.\, \lambda\, \mathsf{x} : \alpha.\, \mathsf{x} \\
[\![\mathbf{Code}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{B}]\!]^{\circ} &\overset{\text{def}}{=} \Pi\, \mathsf{x}' : [\![\mathbf{A}']\!]^{\circ}.\, \Pi\, \mathsf{x} : [\![\mathbf{A}]\!]^{\circ}.\, [\![\mathbf{B}]\!]^{\circ} \\
[\![\boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}]\!]^{\circ} &\overset{\text{def}}{=} \lambda\, \mathsf{x}' : [\![\mathbf{A}']\!]^{\circ}.\, \lambda\, \mathsf{x} : [\![\mathbf{A}]\!]^{\circ}.\, [\![\mathbf{e}]\!]^{\circ} \\
[\![\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle]\!]^{\circ} &\overset{\text{def}}{=} [\![\mathbf{e}]\!]^{\circ}\ \ [\![\mathbf{e}']\!]^{\circ} \\
[\![\mathbf{e}\ \mathbf{e}']\!]^{\circ} &\overset{\text{def}}{=} [\![\mathbf{e}]\!]^{\circ}\ \ [\![\mathbf{e}']\!]^{\circ} \\
&\vdots
\end{aligned}
$$

Figure 5.6: Model of ECC^{CC} in ECC^{D} (excerpts)

I give the new typing rules for ECC^{CC} in Figure 5.5; the full rules are given in Appendix D, Figure D.7 and Figure D.8. Most rules are unchanged from the source language. The most interesting rule is Rule Code, which guarantees that code only type checks when it is closed. This rule captures the goal of typed closure conversion and gives us static machine-checked guarantees that our translation produces closed code. Rule Clo for closures $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ substitutes the environment $\mathbf{e}'$ into the type of the closure, as discussed in Section 5.1. This is similar to Rule App from ECC^{D}, which substitutes a function argument into the result type of a function. As discussed in Section 5.1, this is also critical to type preservation, since the translation must generate closure types with free variables and then synchronize the closure type containing free variables with a closed code type. As with Π types in ECC^{D}, we have two rules for well-typed **Code** types. Rule T-Code-Prop allows impredicativity in **Prop**, while Rule T-Code-Type is predicative.

Note that the impredicative rule, Rule T-Code-Prop, is not necessary for type preservation; we only need to include it if the source language has impredicative functions.

5.2.1 Meta-Theory

I prove type safety and consistency of ECC^{CC} following the standard architecture presented in Chapter 3.

The essence of the model translation from ECC^{CC} to ECC^{D} is given in Figure 5.6. The translation is defined inductively on the syntax of expressions. This includes only the key translation rules. The translation $[\![\mathbf{e}]\!]^{\circ} = \mathsf{e}$ models the ECC^{CC} expression $\mathbf{e}$ as the ECC^{D} expression e. The remaining rules are given in Appendix D, in Figure D.14.[1]

1 In the previous version of this work (Bowman and Ahmed, 2018), this translation was defined by induction on typing derivations. That presentation is verbose and not necessary for constructing a model of ECC^{CC}.

As described in Chapter 3, I always assume I only translate well-typed expressions, so the typing derivation for $\mathbf{e}$ is always an implicit parameter whenever we have $[\![\mathbf{e}]\!]^\circ$.

I model code $\boldsymbol{\lambda}\,\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}.\,\mathbf{e}$ as a curried function $\lambda\,\mathsf{n} : [\![\mathbf{A}']\!]^\circ.\,\lambda\,\mathsf{x} : [\![\mathbf{A}]\!]^\circ.\,[\![\mathbf{e}]\!]^\circ$, and a code type $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{B}$ via the curried dependent function type $\Pi\,\mathsf{n} : [\![\mathbf{A}']\!]^\circ.\,\Pi\,\mathsf{x} : [\![\mathbf{A}]\!]^\circ.\,[\![\mathbf{B}]\!]^\circ$. Observe that the inner function produced in ECC^D is not closed, but that is not a problem since the model only exists to prove type safety and consistency. It is only in ECC^{CC} programs that code must be closed. I model closures $\langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle$ as the application $[\![\mathbf{e}]\!]^\circ\ [\![\mathbf{e}']\!]^\circ$—*i.e.*, the application of the function $[\![\mathbf{e}]\!]^\circ$ to its environment $[\![\mathbf{e}']\!]^\circ$. I model $\langle\rangle$ with the standard Church-encoding as the polymorphic identity function, since ECC^D does not include a unit expression. (We could just as well add a unit expression to ECC^D.) All other rules simply recursively translate subterms.

As discussed in Chapter 3, we first must prove preservation of falseness Lemma 5.2.1. I encode the empty type, or invalid specification, $\bot$ in ECC^{CC} as $\mathbf{\Pi\,A : Prop\,.\,A}$. This type describes a function that takes any arbitrary specification $\mathbf{A}$ and returns a proof of $\mathbf{A}$. There is only a proof of $\bot$ if ECC^{CC} is not consistent. Similar, we encode $\bot$ in ECC^D as $\Pi\,\mathsf{A} : \mathsf{Prop}\,.\,\mathsf{A}$. It is clear from the translation of dependent function types that the translation preserves falseness. I use $=$ as the terms are not just definitionally equivalent, but syntactically identical.

Lemma 5.2.1 (Preservation of Falseness). $[\![\bot]\!]^\circ = \bot$

To prove type preservation, I use the standard architecture from Chapter 3. The proofs are straightforward, since the typing rules in ECC^{CC} essentially correspond to partial application already.

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

Lemma 5.2.2 (Compositionality). $[\![(\mathbf{e}[\mathbf{e}'/\mathbf{x}])]\!]^\circ = [\![\mathbf{e}]\!]^\circ[[\![\mathbf{e}]\!]^\circ/\mathsf{x}]$

Proof. The proof is by induction on the structure of $\mathbf{e}$. I give the key proof cases.

Case: $\mathbf{e} = \mathbf{Prop}$

Trivial, since $\mathbf{e} = \mathbf{Prop}$ cannot have free variables.

Case: $\mathbf{e} = \mathbf{x}'$.

There are two sub-cases:

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

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

Case: $\mathbf{e} = \mathbf{let}\ \mathbf{x} = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2$

Follows easily by the inductive hypotheses, since both the translation of **let** and the definition of substitution are structural, except for the capture avoidance reasoning.

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

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

Case: $\mathbf{e} = \langle\!\langle \mathbf{e_1}, \mathbf{e_2} \rangle\!\rangle$

Must show that $[\![\langle\!\langle \mathbf{e_1}, \mathbf{e_2}\rangle\!\rangle[\mathbf{e'}/\mathbf{x}]]\!]^{\circ} = [\![\langle\!\langle \mathbf{e_1}, \mathbf{e_2}\rangle\!\rangle]\!]^{\circ}[[\![\mathbf{e'}]\!]^{\circ}/\mathbf{x}]$.

$$\begin{aligned}
& [\![\langle\!\langle \mathbf{e_1}, \mathbf{e_2}\rangle\!\rangle[\mathbf{e'}/\mathbf{x}]]\!]^{\circ} \\
= \;& [\![\langle\!\langle \mathbf{e_1}[\mathbf{e'}/\mathbf{x}], \mathbf{e_2}[\mathbf{e'}/\mathbf{x}]\rangle\!\rangle]\!]^{\circ} && (45)\\
& \text{by definition of substitution} \\
= \;& [\![\mathbf{e_1}[\mathbf{e'}/\mathbf{x}]]\!]^{\circ}\ \ [\![\mathbf{e_2}[\mathbf{e'}/\mathbf{x}]]\!]^{\circ} && (46)\\
& \text{by definition of translation} \\
= \;& [\![\mathbf{e_1}]\!]^{\circ}[[\![\mathbf{e'}]\!]^{\circ}/\mathbf{x}]\ [\![\mathbf{e_2}]\!]^{\circ}[[\![\mathbf{e'}]\!]^{\circ}/\mathbf{x}] && (47)\\
& \text{by the induction hypothesis applied to } \mathbf{e_1} \text{ and } \mathbf{e_2} \\
= \;& [\![\langle\!\langle \mathbf{e_1}, \mathbf{e_2}\rangle\!\rangle]\!]^{\circ}[[\![\mathbf{e'}]\!]^{\circ}/\mathbf{x}] && (48)\\
& \text{by substitution}
\end{aligned}$$

□

Next I show that the model preserves reduction, or that our model in ECC^{D} weakly simulates reduction in ECC^{CC}. This is used both to show that equivalence is preserved, since equivalence is defined by conversion, and to show type safety.

Lemma 5.2.3 (Preservation of Reduction)**.** *If* $\mathbf{e} \triangleright \mathbf{e'}$ *then* $[\![\mathbf{e}]\!]^{\circ} \triangleright^{*} [\![\mathbf{e'}]\!]^{\circ}$

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

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

We must show that

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

$$\begin{aligned}
& [\![(\langle\!\langle (\boldsymbol{\lambda}\, \mathbf{x'} : \mathbf{A'}, \mathbf{x} : \mathbf{A}.\, \mathbf{e}_b), \mathbf{e'}\rangle\!\rangle\ \mathbf{e})]\!]^{\circ} \\
= \;& ((\lambda\, x' : [\![\mathbf{A'}]\!]^{\circ}.\, \lambda\, x : [\![\mathbf{A}]\!]^{\circ}.\, [\![\mathbf{e}_b]\!]^{\circ})\ \ [\![\mathbf{e'}]\!]^{\circ})\ \ [\![\mathbf{e}]\!]^{\circ} && \text{by definition} && (49)\\
\triangleright^{2}_{\beta}\; & [\![\mathbf{e}_b]\!]^{\circ}[[\![\mathbf{e'}]\!]^{\circ}/x'][[\![\mathbf{e}]\!]^{\circ}/x] && && (50)\\
= \;& [\![(\mathbf{e}_b[\mathbf{e'}/\mathbf{x'}][\mathbf{e}/\mathbf{x}])]\!]^{\circ} && \text{by Lemma 5.2.2} && (51)
\end{aligned}$$

□

Now I show that conversion is preserved. This essentially follows from preservation of reduction, Lemma 5.2.3.

Lemma 5.2.4 (Preservation of Conversion). *If* $\Gamma \vdash e \rhd^* e'$ *then* $[\![\Gamma]\!]^\circ \vdash [\![e]\!]^\circ \rhd^* [\![e']\!]^\circ$

Proof. The proof is by induction on derivation $\Gamma \vdash e \rhd^* e'$.[2] Each case is essentially uninteresting, but we give a few representative cases.

Case: Rule Red-Refl

Trivial.

Case: Rule Red-Trans

We have that $e \rhd e_1$ and $e_1 \rhd^* e'$. We must show that $[\![e]\!]^\circ \rhd^* [\![e']\!]^\circ$.

By Lemma 5.2.3 applied to $e \rhd^* e_1$, we know that $[\![e]\!]^\circ \rhd^* [\![e_1]\!]^\circ$, and by the induction hypothesis applied to $e_1 \rhd^* e'$ we know that $[\![e_1]\!]^\circ \rhd^* [\![e']\!]^\circ$.

By ECC^D Rule Red-Trans, we conclude that $[\![e]\!]^\circ \rhd^* [\![e']\!]^\circ$.

Case: Rule Red-Cong-Let

We have $$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x = e' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \rhd^* \mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2'}\ \textsc{Red-Cong-Let}$$

We must show that $[\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2]\!]^\circ \rhd^* [\![\mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2']\!]^\circ$, which follows easily by the induction hypothesis and by ECC^D Rule Red-Cong-Let.

Case: Rule Red-Cong-Code

We have $$\frac{\begin{array}{c}\Gamma \vdash A_1 \rhd^* A_1' \\ \Gamma, n : A_1' \vdash A_2 \rhd^* A_2' \qquad \Gamma, n : A_1', x : A_2' \vdash e \rhd^* e'\end{array}}{\Gamma \vdash \boldsymbol{\lambda}\, (n : A_1, x : A_2).\, e \rhd^* \boldsymbol{\lambda}\, (n : A_1', x : A_2').\, e'}\ \textsc{Red-Cong-Code}$$

We must show that $[\![\boldsymbol{\lambda}\, (n : A_1, x : A_2).\, e]\!]^\circ \rhd^* [\![\boldsymbol{\lambda}\, (n : A_1', x : A_2').\, e']\!]^\circ$.

By definition of the translation, we must show that $\lambda\, n : [\![A_1]\!]^\circ.\, \lambda\, x : [\![A_2]\!]^\circ.\, [\![e]\!]^\circ \rhd^* \lambda\, n : [\![A_1']\!]^\circ.\, \lambda\, x : [\![A_2']\!]^\circ\ [\![e']\!]^\circ$..

By the induction hypothesis, we know that

a) $[\![A_1]\!]^\circ \rhd^* [\![A_1']\!]^\circ$

b) $[\![A_2]\!]^\circ \rhd^* [\![A_2']\!]^\circ$

c) $[\![e]\!]^\circ \rhd^* [\![e']\!]^\circ$

The goal follows by two applications of ECC^D Rule Red-Cong-Lam.

Case: Rule Red-Cong-Clo

We must show that $[\![\langle\!\langle e_1, e_2 \rangle\!\rangle]\!]^\circ \rhd^* [\![\langle\!\langle e_1', e_2' \rangle\!\rangle]\!]^\circ$, given that $e_1 \rhd^* e_1'$ and $e_2 \rhd^* e_2'$, which follows easily by the induction hypothesis and by the ECC^D Rule Red-Cong-App.

□

2 In the prior version of this work (Bowman and Ahmed, 2018), this proof was incorrectly stated as by induction on the length of the reduction sequence. This version is corrected.

Next, I show that the translation preserves equivalence. The proof essentially follows from Lemma 5.2.4, but we must show that the η-equivalence for closures is preserved.

Lemma 5.2.5 (Preservation of Equivalence)**.** *If* $\mathbf{e}_1 \equiv \mathbf{e}_2$ *then* $[\![\mathbf{e}_1]\!]^\circ \equiv [\![\mathbf{e}_2]\!]^\circ$

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

Case: Rule $\equiv$ Follows by Lemma 5.2.4 (Preservation of Conversion).

Case: Rule $\equiv$-Clo$_1$

By assumption, we have the following.

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

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

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

We must show that $[\![\mathbf{e}_1]\!]^\circ \equiv [\![\mathbf{e}_2]\!]^\circ$. By Rule $\equiv$-η_1, it suffices to show:

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

$$\begin{aligned}
[\![\mathbf{e}_1]\!]^\circ &\rhd^* [\![(\langle\!\langle \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{e}_1', \mathbf{e}' \rangle\!\rangle)]\!]^\circ && \text{by Lemma 5.2.4} && (52)\\
&= (\lambda\, \mathsf{x}' : [\![\mathbf{A}']\!]^\circ.\, \lambda\, \mathsf{x} : [\![\mathbf{A}]\!]^\circ.\, [\![\mathbf{e}_1']\!]^\circ)\ \ [\![\mathbf{e}']\!]^\circ && && (53)\\
&\rhd \lambda\, \mathsf{x} : [\![\mathbf{A}']\!]^\circ[[\![\mathbf{e}']\!]^\circ/\mathsf{x}'].\, [\![\mathbf{e}_1']\!]^\circ[[\![\mathbf{e}']\!]^\circ/\mathsf{x}'] && && (54)
\end{aligned}$$

b) $[\![\mathbf{e}_2]\!]^\circ \rhd^* [\![\mathbf{e}_2']\!]^\circ$ which follows by Lemma 5.2.4.

c) $[\![\mathbf{e}_1']\!]^\circ[[\![\mathbf{e}']\!]^\circ/\mathsf{x}'] \equiv [\![\mathbf{e}_2']\!]^\circ\ \ \mathsf{x}$, which follows by the inductive hypothesis applied to $\mathbf{e}_1[\mathbf{e}'/\mathbf{x}'] \equiv \mathbf{e}_2'\ \mathbf{x}$ and Lemma 5.2.2.

Case: Rule $\equiv$-Clo$_2$ is symmetric.

□

Lemma 5.2.6 (Preservation of Subtyping)**.** *If* $\boldsymbol{\Gamma} \vdash \mathbf{A} \preceq \mathbf{B}$ *then* $[\![\boldsymbol{\Gamma}]\!]^\circ \vdash [\![\mathbf{A}]\!]^\circ \preceq [\![\mathbf{B}]\!]^\circ$

Proof. The proof is by induction on the derivation of $\boldsymbol{\Gamma} \vdash \mathbf{A} \preceq \mathbf{B}$. All cases are completely uninteresting, but I give a few representative cases anyway.

Case: Rule $\preceq$-$\equiv$

Follows by Lemma 5.2.5.

Case: Rule $\preceq$-Cum

Must show $[\![\mathbf{Type}_i]\!]^\circ \preceq [\![\mathbf{Type}_{i+1}]\!]^\circ$. By translation, we must show that $\mathsf{Type}_i \preceq \mathsf{Type}_{i+1}$ which follows by the ECC^D Rule $\preceq$-Cum.

Case: Rule $\preceq$-Code

Must show $[\![\mathbf{Code}\,(\mathbf{n}_1 : \mathbf{A}_1, \mathbf{x}_1 : \mathbf{A}'_1).\, \mathbf{B}_1]\!]^\circ \preceq [\![\mathbf{Code}\,(\mathbf{n}_2 : \mathbf{A}_2, \mathbf{x}_2 : \mathbf{A}'_2).\, \mathbf{B}_2]\!]^\circ$. By translation, we must show that $\Pi\, \mathsf{n}_1 : [\![\mathbf{A}_1]\!]^\circ.\, \Pi\, \mathsf{x}_1 : [\![\mathbf{A}'_1]\!]^\circ.\, [\![\mathbf{B}_1]\!]^\circ \preceq \Pi\, \mathsf{n}_2 : [\![\mathbf{A}_2]\!]^\circ.\, \Pi\, \mathsf{x}_2 : [\![\mathbf{A}'_2]\!]^\circ.\, [\![\mathbf{B}_2]\!]^\circ$ By Lemma 5.2.5 (Preservation of Equivalence), we know that $[\![\mathbf{A}_1]\!]^\circ \equiv [\![\mathbf{A}'_1]\!]^\circ$ and $[\![\mathbf{A}_2 \equiv \mathbf{A}'_2]\!]^\circ$. By the induction hypothesis, we know that $[\![\mathbf{B}_1]\!]^\circ \preceq [\![\mathbf{B}_2]\!]^\circ$. The goal follows by two applications of the ECC^D Rule $\preceq$-Pi.

□

We can now show the final lemma for type safety and consistency.

Lemma 5.2.7 (Type and Well-formedness Preservation)**.**

1. *If* $\vdash \mathbf{\Gamma}$ *then* $\vdash [\![\mathbf{\Gamma}]\!]^\circ$
2. *If* $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ *then* $[\![\mathbf{\Gamma}]\!]^\circ \vdash [\![\mathbf{e}]\!]^\circ : [\![\mathbf{A}]\!]^\circ$

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

Case: Rule W-Empty

Trivial.

Case: Rule W-Def

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

Case: Rule W-Assum

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

Case: Rule Prop

It suffices to show that $\vdash [\![\mathbf{\Gamma}]\!]^\circ$, since $[\![\mathbf{Prop}]\!]^\circ = \mathsf{Prop}$, which follows by part 1 of the inductive hypothesis.

⋮

Case: Rule T-Code-Prop

We have that

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

We must show that $[\![\mathbf{\Gamma}]\!]^\circ \vdash \Pi\, \mathsf{x}' : [\![\mathbf{A}']\!]^\circ.\, \Pi\, \mathsf{x} : [\![\mathbf{A}]\!]^\circ.\, [\![\mathbf{B}]\!]^\circ : \mathsf{Prop}$

By two applications of Rule Pi-Prop, it suffices to show

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

Case: Rule Code

We have:

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

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

Case: Rule Clo

We have:

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

By definition of the translation, we must show that

$[\![\Gamma]\!]^\circ \vdash [\![e]\!]^\circ \;\; [\![e']\!]^\circ : [\![(\Pi\, x : A[e'/x'].\, B[e'/x'])]\!]^\circ$.

By Lemma 5.2.2 (Compositionality), it suffices to show that

$[\![\Gamma]\!]^\circ \vdash [\![e]\!]^\circ \;\; [\![e']\!]^\circ : \Pi x : [\![A]\!]^\circ[[\![e']\!]^\circ/x'].\, [\![B]\!]^\circ[[\![e']\!]^\circ/x']$.

By Rule App in ECC^D, it suffices to show that

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

Case: Rule App

Similar to the case for Rule Clo.

Case: Rule Conv

Follows by part 2 of the inductive hypothesis and Lemma 5.2.6 (Preservation of Subtyping).

□

The model of ECC^{CC} in ECC^D implies the desired consistency and type safety theorems, as discussed earlier.

Consistency tells us that we can only write proofs of valid specifications.

Theorem 5.2.8 (Logical Consistency of ECC^{CC})**.** *There does not exist a closed expression* e *such that* $\cdot \vdash e : \bot$.

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

$$
\begin{array}{rcl}
[\![x]\!] & \overset{\text{def}}{=} & \mathbf{x} \\
[\![\mathsf{Prop}]\!] & \overset{\text{def}}{=} & \mathbf{Prop} \\
[\![\mathsf{Type}_i]\!] & \overset{\text{def}}{=} & \mathbf{Type}_i \\
[\![\Pi\, x : A.\, B]\!] & \overset{\text{def}}{=} & \mathbf{\Pi\, x} : [\![A]\!].\, [\![B]\!] \\
[\![\lambda\, x : A.\, e]\!] & \overset{\text{def}}{=} & \langle\!\langle (\boldsymbol{\lambda}\, (\mathbf{n} : \mathbf{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \ldots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]). \\
& & \quad\quad \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![e]\!]), \\
& & \quad \langle \mathbf{x}_i \ldots \rangle\ \mathbf{as}\ \mathbf{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \ldots) \rangle\!\rangle \\
& & x_i : A_i \ldots\ = \mathrm{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma) \\
[\![e_1\ e_2]\!] & \overset{\text{def}}{=} & [\![e_1]\!]\ [\![e_2]\!] \\
& \vdots & \\
[\![\mathsf{let}\, x = e\ \mathsf{in}\ e']\!] & \overset{\text{def}}{=} & \mathbf{let}\ \mathbf{x} = [\![e]\!]\ \mathbf{in}\ [\![e']\!]
\end{array}
$$

Figure 5.7: Abstract Closure Conversion from ECC^D to ECC^{CC} (excerpts)

Type safety tells us that there is no undefined behavior that causes a program to get stuck before it produces an observation.

Theorem 5.2.9 (Type Safety of ECC^{CC}). *If* $\vdash \mathbf{e}$, *then* $\mathbf{eval}(\mathbf{e})$ *is well-defined.*

5.3 Closure Conversion

I present the key closure conversion translation rules in Figure 5.7. Formally, the translation is defined by induction on typing derivations. This is necessary since the translation must produce a type annotation for the environment argument of the code. The translation $[\![e]\!]$ takes the typing derivation for e as an implicit parameter. For concision, I give a complete definition of translation explicitly defined over typing derivations in Appendix E, Figure E.4 and Figure E.5.

Every case of the translation except for functions is trivial, including application since application is still the elimination form for closures after closure conversion. In the non-trivial case, we translate ECC^D dependent functions to ECC^{CC} closures, as described in Section 5.1. The translation of a function $[\![\lambda\, x : A.\, e]\!]$ produces a closure $\langle\!\langle \mathbf{e}_1, \mathbf{e}_2 \rangle\!\rangle$. The first component $\mathbf{e}_1$ is closed code. Ignoring the type annotation for a moment, the code $\boldsymbol{\lambda}\, (\mathbf{n}, \mathbf{x}).\, \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![e]\!]$ projects each of the $|i|$ free variables $\mathbf{x}_i \ldots$ from the environment $\mathbf{n}$ and binds them in the scope of the body $[\![e]\!]$. Since ECC^D and ECC^{CC} are dependently typed, we must also bind the free variables from the environment in the type annotation for the argument $\mathbf{x}$, *i.e.*, producing the annotation $\mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]$ instead of just $\mathbf{x} : [\![A]\!]$. Next we produce the environment type $\mathbf{\Sigma}\, (\mathbf{x}_i : [\![A]\!] \ldots)$, from the free source variables $x_i \ldots$ of types $A_i \ldots$. We create the

$$\begin{aligned}
\mathrm{FV}(\mathsf{e}, \mathsf{B}, \Gamma) \quad & \stackrel{\text{def}}{=} \quad \Gamma_0, \ldots, \Gamma_n, (\mathsf{x}_0 : \mathsf{A}_0, \ldots, \mathsf{x}_n : \mathsf{A}_n) \\
\textit{where} \quad & \mathsf{x}_0, \ldots, \mathsf{x}_n = \mathrm{fv}(\mathsf{e}, \mathsf{B}) \\
& \Gamma \vdash \mathsf{x}_0 : \mathsf{A}_0 \\
& \quad \vdots \\
& \Gamma \vdash \mathsf{x}_n : \mathsf{A}_n \\
& \Gamma_0 = \mathrm{FV}(\mathsf{A}_0, \_, \Gamma) \\
& \quad \vdots \\
& \Gamma_n = \mathrm{FV}(\mathsf{A}_n, \_, \Gamma)
\end{aligned}$$

Figure 5.8: Dependent Free Variable Sequences

environment $\mathbf{e}_2$ by creating the dependent n-tuple $\langle \mathbf{x}_i \ldots \rangle$; these free variables will be instantiated with values at run time before calling the closure.

Notice that application is translated to application. In abstract closure conversion, the closure is not a pair; its elimination form is still just application, as shown in Figure 5.2. This makes the translation of application deceptively simple compared to other closure conversion translations.

Computing free variables in a dependently typed language is more complex than usual. To compute the sequence of free variables and their types, I define the metafunction $\mathrm{FV}(\mathsf{e}, \mathsf{B}, \Gamma)$ in Figure 5.8. Just from the syntax of terms e, B, we can compute some sequence of free variables $\mathsf{x}_0, \ldots, \mathsf{x}_n = \mathrm{fv}(\mathsf{e}, \mathsf{B})$. However, the types of these free variables $\mathsf{A}_0, \ldots, \mathsf{A}_n$ may contain *other* free variables, and their types may contain still others, and so on! We must, therefore, recursively compute the sequence of free variables and their types with respect to a typing environment Γ. Note that because the type B of a term e may contain different free variables than the term, we must compute the sequence with respect to both a term and its type. However, in all recursive applications of this metafunction—*e.g.*, $\mathrm{FV}(\mathsf{A}_0, \_, \Gamma)$—the type of A_0 must be a universe and cannot have any free variables.

5.3.1 Type Preservation

I prove type preservation, using the standard architecture from Chapter 3.

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

Lemma 5.3.1 (Compositionality). $[\![(e_1[e_2/x])]\!] \equiv [\![e_1]\!][[\![e_2]\!]/x]$

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

Case: Rule Var

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

Case: Rule Pi-Prop

We know that $e_1 = \Pi\, x' : A.\, B$. W.l.o.g., assume $x' \neq x$.

We must show $[\![(\Pi\, x' : A[e_2/x].\, B[e_2/x])]\!] \equiv [\![(\Pi\, x' : A.\, B)]\!][[\![e_2]\!]/\mathbf{x}]$.

$$\begin{aligned}
& [\![(\Pi\, x' : A[e_2/x].\, B[e_2/x])]\!] && (55)\\
= {} & \mathbf{\Pi\, x'} : [\![(A[e_2/x])]\!].\, [\![(B[e_2/x])]\!] && (56)\\
& \text{by definition of the translation} \\
= {} & \mathbf{\Pi\, x'} : ([\![A]\!][[\![e_2]\!]/\mathbf{x}]).\, ([\![B]\!][[\![e_2]\!]/\mathbf{x}]) && (57)\\
& \text{by the inductive hypothesis for } A \text{ and } B \\
= {} & (\mathbf{\Pi\, x'} : [\![A]\!].\, [\![B]\!])[[\![e_2]\!]/\mathbf{x}] && (58)\\
& \text{by definition of substitution} \\
= {} & [\![(\Pi\, x' : A.\, B)]\!][[\![e_2]\!]/\mathbf{x}] && (59)\\
& \text{by definition of translation}
\end{aligned}$$

Case: Rule Pi-Type. Similar to Rule Pi-Prop

Case: Rule Lam

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

$$\begin{aligned}
& [\![((\lambda\, y : A.\, e)[e_2/x])]\!] && (60)\\
= {} & [\![(\lambda\, y : (A[e_2/x]).\, e[e_2/x])]\!] && (61)\\
& \text{by substitution} \\
= {} & \langle\!\langle (\boldsymbol{\lambda}\, \mathbf{n} : \mathbf{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \ldots), \mathbf{y} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\, \mathbf{in}\, [\![(A[e_2/x])]\!]. && (62)\\
& \quad \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\, \mathbf{in}\, [\![(e[e_2/x])]\!]), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle \\
& \text{by definition of the translation}
\end{aligned}$$

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

On the other hand, we have

$$\begin{aligned}
\mathbf{f} &= [\![(\lambda\, y : A.\, e)]\!][[\![e_2]\!]/x] && (63)\\
&= \langle\!\langle(\boldsymbol{\lambda}\, \mathbf{n} : \boldsymbol{\Sigma}\,(\mathbf{x}_j : [\![A_j]\!] \ldots), \mathbf{y} : \mathbf{let}\ \langle \mathbf{x}_j \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]. && (64)\\
&\qquad \mathbf{let}\ \langle \mathbf{x}_j \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![e]\!]), \langle \mathbf{x}_{j_0} \ldots, [\![e_2]\!], \mathbf{x}_{j_{i+1}} \ldots\rangle\rangle\!\rangle\\
&\quad \text{by definition of the translation}
\end{aligned}$$

where $x_j : A_j \ldots = \mathrm{FV}(\lambda\, y : A.\, e, \Gamma)$. Note that x is in $x_j \ldots$; we can write the sequence as $(x_{j_0} \ldots x, x_{j_{i+1}} \ldots)$. Therefore, the environment we generate contains $[\![e_2]\!]$ in position j_i.

By Rule $\equiv$-Clo$_1$, it suffices to show that

$$\mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \langle \mathbf{x}_i \ldots\rangle\ \mathbf{in}\ [\![(e[e_2/x])]\!] \equiv \mathbf{f}\ \mathbf{y}$$

where $\mathbf{f}$ is the closure from Equation (63).

$$\begin{aligned}
\mathbf{f}\ \mathbf{y} &\equiv \mathbf{let}\ \langle \mathbf{x}_{j_0} \ldots \mathbf{x}, \mathbf{x}_{j_{i+1}\ldots}\rangle = \langle \mathbf{x}_{j_0} \ldots, [\![e_2]\!], \mathbf{x}_{j_{i+1}} \ldots\rangle\ \mathbf{in}\ [\![e]\!] && (65)\\
&\quad \text{by } \rhd_\beta \text{ in ECC}^{CC}\\
&\equiv [\![e]\!][[\![e_2]\!]/\mathbf{x}] && (66)\\
&\quad \text{by } |\mathrm{j}| \text{ applications of } \rhd_\zeta\text{, since only } \mathbf{x} \text{ has a value}\\
&\equiv [\![(e[e_2/x])]\!] && (67)\\
&\quad \text{by the inductive hypothesis applied to the derivation for } e\\
&\equiv \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \langle \mathbf{x}_i \ldots\rangle\ \mathbf{in}\ [\![(e[e_2/x])]\!] && (68)\\
&\quad \text{by } |\mathrm{i}| \text{ applications of } \rhd_\zeta\text{, since no variable has a value}
\end{aligned}$$

□

Next I show preservation of reduction: if a source term e takes a step, then its translation $[\![e]\!]$ is convertible to a definitionally equivalent term $\mathbf{e}$. This proof essentially follows by Lemma 5.3.1. Note that since Lemma 5.3.1 relies on our η-equivalence rule for closures, we can only show reduction up to definitional equivalence. That is, we cannot show $[\![e]\!] \rhd^* [\![e']\!]$. This is not a problem; we reason about source programs to equivalence anyway, and not up to syntactic equality.

Lemma 5.3.2 (Preservation of Reduction). *If* $\Gamma \vdash e \rhd e'$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] \rhd^* \mathbf{e}$ *and* $\mathbf{e} \equiv [\![e']\!]$

Proof. By cases on $\Gamma \vdash e \rhd e'$. Most cases follow easily by Lemma 5.3.1, since most cases of reduction are defined by substitution.

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

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

Case: $\mathsf{let}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2 \rhd_\zeta \mathsf{e}_2[\mathsf{e}_1/\mathsf{x}]$

We must show that $[\![(\mathsf{let}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2)]\!] \rhd^* \mathbf{e}$ and $[\![(\mathsf{e}_2[\mathsf{e}_1/\mathsf{x}])]\!] \equiv \mathbf{e}$. Let $\mathbf{e} \overset{\text{def}}{=} [\![\mathsf{e}_2]\!][[\![\mathsf{e}_1]\!]/\mathbf{x}]$.

$$
\begin{aligned}
[\![(\mathsf{let}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2)]\!] &= \mathbf{let}\,\mathbf{x} = [\![\mathsf{e}_1]\!]\ \mathbf{in}\ [\![\mathsf{e}_2]\!] && (69)\\
&\quad \text{by definition of the translation}\\
&\rhd_\zeta [\![\mathsf{e}_2]\!][[\![\mathsf{e}_1]\!]/\mathbf{x}] && (70)\\
&\equiv [\![(\mathsf{e}_2[\mathsf{e}_1/\mathsf{x}])]\!] && (71)\\
&\quad \text{by Lemma 5.3.1 (Compositionality)}
\end{aligned}
$$

Case: $(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1)\ \mathsf{e}_2 \rhd_\beta \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]$

We must show that $[\![((\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1)\ \mathsf{e}_2)]\!] \rhd^* \mathbf{e}$ and $[\![(\mathsf{e}_2[\mathsf{e}_1/\mathsf{x}])]\!] \equiv \mathbf{e}$. Let $\mathbf{e} \overset{\text{def}}{=} [\![\mathsf{e}_1]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}]$.

By definition of the translation, $[\![((\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1)\ \mathsf{e}_2)]\!] = \mathbf{f}\ \ [\![\mathsf{e}_2]\!]$, where

$$
\begin{aligned}
\mathbf{f} = \langle\!\langle(\boldsymbol{\lambda}\,\mathbf{n} : \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots), \mathbf{x} : \mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\,\mathbf{in}\ [\![\mathsf{A}]\!].\\
\mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\,\mathbf{in}\ [\![\mathsf{e}_1]\!]), \langle \mathbf{x}_i \ldots\rangle\rangle\!\rangle
\end{aligned}
$$

and where $\mathsf{x}_i : \mathsf{A}_i \ldots = \mathrm{FV}(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}_1, \Gamma)$.

To complete the proof, observe that,

$$
\begin{aligned}
\mathbf{f}\ \ [\![\mathsf{e}_2]\!] &\rhd_\beta \mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \langle \mathbf{x}_i \ldots\rangle\ \mathbf{in}\,[\![\mathsf{e}_1]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}] && (72)\\
&\rhd_\zeta^{|i|} [\![\mathsf{e}_1]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}] && (73)\\
&\equiv [\![(\mathsf{e}_1[\mathsf{e}_2/\mathsf{x}])]\!] \qquad \text{by Lemma 5.3.1} && (74)
\end{aligned}
$$

□

Lemma 5.3.3 (Preservation of Conversion)**.** *If* $\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \rhd^* \mathbf{e}$ *and* $[\![\Gamma]\!] \vdash \mathbf{e} \equiv [\![\mathsf{e}']\!]$.

Proof. By induction the derivation $\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'$.[3] I give the key proof cases.

Case: Rule Red-Refl

Trivial.

Case: Rule Red-Trans

Follows by Lemma 5.3.2 and the induction hypothesis.

3 In the previous version of this work (Bowman and Ahmed, 2018), this proof was incorrectly stated by induction on the length of the reduction sequence.

Case: Rule Red-Cong-Let

Follows by Rule Red-Cong-Let and the induction hypothesis.

Case: Rule Red-Cong-App

Follows by Rule Red-Cong-App and the induction hypothesis.

Case: Rule Red-Cong-Lam

We have that $\dfrac{\Gamma \vdash A \rhd^* A' \qquad \Gamma, x : A' \vdash e \rhd^* e'}{\Gamma \vdash \lambda x : A.\, e \rhd^* \lambda x : A'.\, e'}$ Red-Cong-Lam

We must show that $[\![\Gamma]\!] \vdash [\![\lambda x : A.\, e]\!] \rhd^* \mathbf{e}_m$ and $\mathbf{e}_m \equiv [\![\lambda x : A'.\, e']\!]$

By definition of the translation, it suffices to show that the following is convertible to $\mathbf{e}_m \equiv [\![\lambda x : A'.\, e']\!]$.

$$\begin{aligned}[\![\lambda x : A.\, e]\!] = \langle\!\langle (\boldsymbol{\lambda}\, (\mathbf{n} : \boldsymbol{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \dots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]).\\ \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ [\![e]\!]),\\ \langle \mathbf{x}_i \dots \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \dots)\rangle\!\rangle\end{aligned}$$

where $x_i : A_i \dots\ = \mathrm{FV}(\lambda x : A.\, e, \Pi x : A.\, B, \Gamma)$

By the induction hypothesis applied to $A \rhd^* A'$ and $e \rhd^* e'$, we know that $[\![A]\!] \rhd^* \mathbf{A} \equiv [\![A']\!]$ and $[\![e]\!] \rhd^* \mathbf{e} \equiv [\![e']\!]$.

Therefore, by Rule Red-Cong-Clo, Rule Red-Cong-Code, and Rule Red-Cong-Let, we know

$$\begin{aligned}[\![\lambda x : A.\, e]\!] \rhd^* \langle\!\langle (\boldsymbol{\lambda}\, (\mathbf{n} : \boldsymbol{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \dots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{A}).\\ \mathbf{let}\ \langle \mathbf{x}_i \dots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e}),\\ \langle \mathbf{x}_i \dots \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, (\mathbf{x}_i : [\![A_i]\!] \dots)\rangle\!\rangle\end{aligned}$$

We must show this term is equivalent to $[\![\lambda x : A'.\, e']\!]$.

By Rule ≡-Clo1, it suffices to show that the bodies of the closures are equivalent, *i.e.*, that $\mathbf{e} \equiv [\![e']\!]$, which we know by the earlier appeal to the induction hypothesis applied to $e \rhd^* e'$.

□

I next prove equivalence preservation. As equivalence is defined primarily by conversion, the only interesting part of the next proof is preserving η-equivalence. To show that η-equivalence is preserved, we require the new η rules for closures.

Lemma 5.3.4 (Preservation of Equivalence)**.** *If* $\Gamma \vdash e \equiv e'$, *then* $[\![\Gamma]\!] \vdash [\![e]\!] \equiv [\![e']\!]$.

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

Case: Rule $\equiv$

By assumption, $\mathsf{e} \triangleright^* \mathsf{e}_1$ and $\mathsf{e}' \triangleright^* \mathsf{e}_1$.

By Lemma 5.3.3, $[\![\mathsf{e}]\!] \triangleright^* \mathbf{e}$ and $\mathbf{e} \equiv [\![\mathsf{e}_1]\!]$, and similarly. $[\![\mathsf{e}']\!] \triangleright^* \mathbf{e}'$ and $\mathbf{e}' \equiv [\![\mathsf{e}_1]\!]$. The result follows by symmetry and transitivity.

Case: Rule $\equiv$-η_1

By assumption, $\mathsf{e} \triangleright^* \lambda\, \mathsf{x} : \mathsf{t}.\, \mathsf{e}_1$, $\mathsf{e}' \triangleright^* \mathsf{e}_2$ and $\mathsf{e}_1 \equiv \mathsf{e}_2\ \mathsf{x}$.

Must show $[\![\mathsf{e}]\!] \equiv [\![\mathsf{e}']\!]$.

By Lemma 5.3.3, $[\![\mathsf{e}]\!] \triangleright^* \mathbf{e}$ and $\mathbf{e} \equiv [\![(\lambda\, \mathsf{x} : \mathsf{t}.\, \mathsf{e}_1)]\!]$, and similarly $[\![\mathsf{e}']\!] \triangleright^* \mathbf{e}'$ and $\mathbf{e}' \equiv [\![\mathsf{e}_2]\!]$.

By transitivity of $\equiv$, it suffices to show $[\![(\lambda\, \mathsf{x} : \mathsf{t}.\, \mathsf{e}_1)]\!] \equiv [\![\mathsf{e}_2]\!]$.

By definition of the translation,

$$[\![(\lambda\, \mathsf{x} : \mathsf{t}.\, \mathsf{e}_1)]\!] = \langle\!\langle (\boldsymbol{\lambda}\, \mathbf{n} : \boldsymbol{\Sigma}\, (\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{A}]\!].$$
$$\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{e}_1]\!]), \langle \mathbf{x}_i \ldots \rangle \rangle\!\rangle$$

where $\mathsf{x}_i : \mathsf{A}_i \ldots = \mathrm{FV}(\lambda\, \mathsf{x} : \mathsf{t}.\, \mathsf{e}_1, \Gamma)$.

By Rule $\equiv$-CLO$_1$ in ECC^{CC}, it suffices to show that

$$\begin{aligned} & \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ [\![\mathsf{e}_1]\!] \\ \equiv\ & [\![\mathsf{e}_1]\!] && (75) \\ & \text{by } |\mathrm{i}| \text{ applications of } \triangleright_\zeta \\ \equiv\ & [\![\mathsf{e}_2]\!]\ \mathbf{x} && (76) \\ & \text{by the inductive hypothesis applied to } \mathsf{e}_1 \equiv \mathsf{e}_2\ \mathsf{x} \end{aligned}$$

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

□

Lemma 5.3.5 (Preservation of Subtyping). *If* $\Gamma \vdash \mathsf{A} \preceq \mathsf{B}$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{A}]\!] \preceq [\![\mathsf{B}]\!]$

Proof. By induction on the derivation $\Gamma \vdash \mathsf{A} \preceq \mathsf{B}$. I give the key cases.

Case: Rule $\preceq$-$\equiv$

We know that $\mathsf{A} \equiv \mathsf{B}$, and must show that $[\![\mathsf{A}]\!] \preceq [\![\mathsf{B}]\!]$. By Rule $\preceq$-$\equiv$, it suffices to show that $[\![\mathsf{A}]\!] \equiv [\![\mathsf{B}]\!]$ which follows by Lemma 5.3.4 (Preservation of Equivalence) and transitivity of $\equiv$.

Case: Rule $\preceq$-CUM

We know that $\mathsf{Type}_i \preceq \mathsf{Type}_{i+1}$, and we must show that $[\![\mathsf{Type}_i]\!] \preceq [\![\mathsf{Type}_{i+1}]\!]$, which follows trivially from the definition of the translation and Rule $\preceq$-CUM.

Case: Rule $\preceq$-Pi

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

We must show that $[\![\Pi\, x_1 : A_1.\, B_1]\!] \preceq [\![\Pi\, x_2 : A_2.\, B_2]\!]$.

By the definition of the translation, it suffices to show that $\mathbf{\Pi\, x_1} : [\![A_1]\!].\, [\![B_1]\!] \preceq \mathbf{\Pi\, x_2} : [\![A_2]\!].\, [\![B_2]\!]$.

By Rule $\preceq$-Pi, it suffices to show that

a) $[\![A_1]\!] \equiv [\![A_2]\!]$, which follows by Lemma 5.3.4, and

b) $[\![B_1]\!] \preceq [\![B_2]\!]$, which follows by the induction hypothesis applied to $B_1 \preceq B_2$.

□

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

Lemma 5.3.6 (Type and Well-formedness Preservation)**.**

1. *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]$

2. *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$

Proof. Parts 1 and 2 proven by simultaneous induction on the mutually defined judgments $\vdash \Gamma$ and $\Gamma \vdash e : A$.

Part 1 follows easily by the induction hypotheses. I give the key cases for part 2.

Case: Rule Lam

We have that $\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B$. We must show that $[\![\Gamma]\!] \vdash [\![(\lambda\, x : A.\, e)]\!] : [\![(\Pi\, x : A.\, B)]\!]$.

By definition of the translation, we must show that

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

has type $\mathbf{\Pi\, x} : [\![A]\!].\, [\![B]\!]$, where $x_i : A_i \ldots\ = \mathrm{FV}(\lambda\, x : t.\, e_1, \Gamma)$.

Notice that the annotation in the term $\mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]$, does not match the annotation in the type $\mathbf{x} : [\![A]\!]$. However, by Rule Clo, we can derive that the closure has type:

$$\mathbf{\Pi}\,(\mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \langle \mathbf{x}_i \ldots\rangle\ \mathbf{in}\ [\![A]\!]).\,(\mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \langle \mathbf{x}_i \ldots\rangle\ \mathbf{in}\ [\![B]\!]),$$

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

First note that $\Gamma \vdash e : A$ implies $\vdash \Gamma$ (Luo, 1990). By part 1 of the induction hypothesis applied to $\vdash \Gamma$, we know $\vdash [\![\Gamma]\!]$. Since each of $x_i : A_i \ldots$ come from Γ, and $[\![\Gamma]\!]$ is well-formed, we know each type in $[\![\Gamma]\!]$ is well-typed. Thus the following explicit environment constructed by closure conversion is well-typed: $[\![\Gamma]\!] \vdash \langle \mathbf{x}_i \ldots \rangle : \mathbf{\Sigma}\,(\mathbf{x}_i : [\![A_i]\!] \ldots)$.

Now we must show that the code

$$\begin{aligned}(\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{\Sigma}\,(\mathbf{x}_i : [\![A_i]\!] \ldots), \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]).\\ \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![e_1]\!])\end{aligned}$$

has type $\mathbf{Code}\,(\mathbf{n}, \mathbf{x}).\,\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![B]\!]$. For brevity, I omit the duplicate type annotations on $\mathbf{n}$ and $\mathbf{x}$.

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

$\mathbf{n} : \mathbf{\Sigma}\,(\mathbf{x}_i : [\![A_i]\!] \ldots) \vdash \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!] : [\![U]\!]$.

Hence, by Rule Code, it suffices to show

$\cdot, \mathbf{n}, \mathbf{x} \vdash \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![e_1]\!] : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![B]\!]$

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

Case: Rule App

We have that $\Gamma \vdash e_1\ e_2 : B[e_2/x]$. We must show that $[\![\Gamma]\!] \vdash [\![e_1]\!]\ \ [\![e_2]\!] : [\![(B[e_2/x])]\!]$. By Lemma 5.3.1, it suffices to show $[\![\Gamma]\!] \vdash [\![e_1]\!]\ \ [\![e_2]\!] : [\![B]\!][[\![e_2]\!]/\mathbf{x}]$, which follows by Rule App and the inductive hypothesis applied to e_1, e_2 and B. □

Theorem 5.3.7 (Type Preservation). *If* $\Gamma \vdash e : t$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![t]\!]$.

5.3.2 Compiler Correctness

Now I prove correctness of separate compilation. Unlike in Chapter 4, there are no secondary evaluation semantics, so the proof follows easily following the standard architecture from Chapter 3.

As usual, I define linking as substitution and use the standard cross-language relation Chapter 2. Components in both ECC^D and ECC^{CC} are well-typed open terms, *i.e.*, $\Gamma \vdash e : A$. I extend the compiler to closing substitutions $[\![\gamma]\!]$ by point-wise application of the translation.

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

Theorem 5.3.8 (Separate Compilation Correctness). *If* $\Gamma \vdash e$*,* $\Gamma \vdash \gamma$*,* $[\![\Gamma]\!] \vdash \boldsymbol{\gamma}$*, and* $[\![\gamma]\!] \equiv \boldsymbol{\gamma}$ *then* $\mathsf{eval}(\gamma(e)) \approx \mathbf{eval}(\boldsymbol{\gamma}([\![e]\!]))$

$$
\begin{array}{llll}
\textit{Values} & \mathbf{V} & ::= & \mathbf{x} \mid .... \mid \mathbf{Code}\,(\mathbf{n} : \mathbf{A'}, \mathbf{x} : \mathbf{A}).\,\mathbf{B} \mid \langle\langle \mathbf{V}, \mathbf{V} \rangle\rangle \\
 & & \mid & \mathbf{\Pi\, x : A.\, B} \mid \boldsymbol{\lambda}\,\mathbf{n} : \mathbf{A'}, \mathbf{x} : \mathbf{A}.\,\mathbf{M} \\
\textit{Computations} & \mathbf{N} & ::= & \mathbf{V} \mid .... \mid \mathbf{V}\ \mathbf{V} \\
\textit{Configurations} & \mathbf{M}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{N} \mid \mathbf{let\, x = N\, in\, M}
\end{array}
$$

Figure 5.9: ECC^{CC} ANF (excerpts)

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

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

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

As a simple corollary, the compiler must also be whole-program correct.

Corollary 5.3.9 (Whole-Program Correctness)**.** *If* $\vdash \mathsf{e}$ *then* $\mathsf{eval}(\mathsf{e}) \approx \mathbf{eval}(\llbracket\mathsf{e}\rrbracket)$.

5.3.3 ANF Preservation

Ultimately we want to compose ANF and closure conversion, so we need closure conversion to preserve ANF. For simplicity, I've defined a more general closure conversion over ECC^D terms, but the translation also preserves ANF.

Typographical Note. *In this section, I restrict the source language from ECC^D to ECC^A. ECC^A is a source language in this section, so I typeset it in* blue, non-bold, sans-serif font.

I define the key ANF syntax for ECC^{CC} in Figure 5.9; the full figure is given in Appendix D Figure D.15. Recall from Chapter 4 that we define configurations, computations, and values in ANF. The only interesting addition is closures, which are values and must contain values as components. Recall that I have an incomplete ANF translation for dependent conditionals in Chapter 4, so I exclude dependent conditionals from the ANF definitions here.

To show ANF is preserved, we must show that configurations are translated to configurations, computations to computations, and values to values. Because the syntactic categories are mutually defined, we must show each of these is preserved simultaneously.

Theorem 5.3.10 (Preservation of ANF)**.** *Let* $\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A}$*;*

1. *If* e *is* V *then* $\llbracket\mathsf{V}\rrbracket = \mathbf{V}$.

2. *If* e *is* N *then* $[\![\mathsf{N}]\!] = \mathbf{N}$.

3. *If* e *is* M *then* $[\![\mathsf{M}]\!] = \mathbf{M}$.

Proof. Formally, the proof is by induction on the typing derivation for the expression e being translated, assuming e is in ANF, *i.e.*, is either a V, an N or an M. The typing derivation is only necessary since the translation must be defined on typing derivations, as discussed in Chapter 3. For simplicity, I present the proof as over the syntax of the expression e, but note that in the case of functions we need the induction hypothesis for a sub-derivation rather than a sub-expression. I give the key cases.

Case: $\mathsf{e} = \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ Note that e is a value V, A and B are configurations, and the translation $\mathbf{e} = \boldsymbol{\Pi}\,\mathbf{x} : [\![\mathsf{A}]\!].\,\mathsf{B}$. By part 3 of the induction hypothesis, $[\![\mathsf{A}]\!]$ and $[\![\mathsf{B}]\!]$ are configurations, so $\mathbf{e}$ is a value.

Case: $\mathsf{e} = \mathsf{let}\,\mathsf{x} = \mathsf{N}\,\mathsf{in}\,\mathsf{M}'$ Note that e is a configuration. We must show that the translation $\mathbf{e} = \mathbf{let}\,\mathbf{x} = [\![\mathsf{N}]\!]\ \mathbf{in}\ [\![\mathsf{M}']\!]$ is a configuration, which follows by parts 2 and 3 of the induction hypothesis.

Case: $\mathsf{e} = \lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{M}$ Note that e is a value. We must show that the translation $\mathbf{e}$, defined as follows, is a value.

$$\begin{aligned}\mathbf{e} = \langle\!\langle (&\boldsymbol{\lambda}\,(\mathbf{n} : \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots), \mathbf{x} : \mathbf{let}\,\langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\,\mathbf{in}\ [\![\mathsf{A}]\!]).\\ &\quad \mathbf{let}\,\langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\,\mathbf{in}\ [\![\mathsf{e}]\!]),\\ &\langle \mathbf{x}_i \ldots \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots)\rangle\!\rangle\end{aligned}$$

where $\mathsf{x}_i : \mathsf{A}_i \ldots = \mathrm{FV}(\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}, \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}, \Gamma)$

It suffices to show that both the code and environment are values.

The environment $\langle \mathbf{x}_i \ldots \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots)$ is a value if $[\![\mathsf{A}_i]\!] \ldots$ are configurations, which is true by part 3 of the induction hypothesis applied to the typing derivations for $\mathsf{A}_i \ldots$ (which are sub-derivations implied by the well-typedness of e).

The code is a value if

a) $\boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![\mathsf{A}_i]\!] \ldots)$ is a configuration, which is true by part 3 of the induction hypothesis applied to the typing derivations for $\mathsf{A}_i \ldots$.

b) $\mathbf{let}\,\langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\,\mathbf{in}\ [\![\mathsf{A}]\!]$ is a configuration, which is true by part 3 of the induction hypothesis applied to A.

c) $\mathbf{let}\,\langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\,\mathbf{in}\ [\![\mathsf{e}]\!]$ is a configuration, which is true by part 3 of the induction hypothesis applied to $[\![\mathsf{e}]\!]$.

Case: $\mathsf{e} = \mathsf{V}\ \mathsf{V}'$ We must show that $\mathbf{e} = \mathbf{V}\ \mathbf{V}'$, which follows by part 1 of the induction hypothesis.

□

6 CONTINUATION-PASSING STYLE

In this chapter, I develop a type-preserving CPS translation for a subset of ECC^D. As mentioned in Chapter 4, CPS presents many challenges and does not scale well to all features of dependency. However, by understanding the key problems that CPS introduces into a dependent types system, we can develop type preserving CPS for some dependently typed languages, which is surprising given past impossibility results.

I start with a brief discussion of the different uses of CPS and discuss the past impossibility result for dependent-type-preserving CPS. I then develop CoC^D, a restriction of ECC^D suitable for translations that rely on *parametricity*, before developing both CBN and CBV type-preserving CPS translations for CoC^D.

Typographical Note. *In this chapter, I typeset the source language, CoC^D, in a* blue, non-bold, sans-serif font*, and the target language, $\mathbf{CoC}^k$, in a* **bold, red, serif font***.*

6.1 On CPS

The CPS translation presented in this chapter allows implementing type-preserving compilation for some dependently typed languages, but it will not allow implementing control effects, a common use for CPS. Many of the design decisions I make do not apply when implementing control effects, or specifically disallow implementing control effects. For example, my goal is to avoid restricting dependencies in the source language so that I can compile existing languages such as Coq. However, to allow mixing control effects and dependency, one must necessarily restrict certain dependencies, at least for the effectful parts of the language. This requires different designs for the CPS language and translation; I discuss some related work in this vein in Section 6.7. Before I present my CPS translation, I discuss the context of CPS translation as it applies to type preservation and the core features of dependency.

Typically, type-preserving compilers use one of two common type translations: (1) the *double-negation translation* that translates expressions of type A into a computation of type $A^{\div} = (A^{+} \to \bot) \to \bot$ or $\neg\neg A^{+}$ where A^{+} represents the value-type translation, or (2) the *locally polymorphic answer type translation* that translates expressions of type A into computations of type $A^{\div} = \forall \alpha.(A^{+} \to \alpha) \to \alpha$. The value translation differs depending on whether the translation is encoding CBN or CBV evaluation order, but essentially recursively translates types that must be values using the value translation $^{+}$, and types that could be computations using the computation translation $^{\div}$. In each translation, the computation expects a continuation, either of type $A^{+} \to \bot$ or $A^{+} \to \alpha$. Intuitively, the computation is forced to call that continuation with a value of type A^{+}.

The double-negation translation represents the final result of a program with the empty type $\bot$ to indicate that programs *never return*, *i.e.*, programs no longer behave like mathematical functions that take inputs and produce output values but instead run to completion and end up in some final answer state. The locally polymorphic answer type uses parametricity to encode a similar idea: if the computation behaves parametrically in its answer type, then no computation can do anything but call its continuation with the *underlying value* that the computation represents. At the level of a whole program, this is equivalent to a program never returning—each intermediate computation cannot return because it must, by parametricity, invoke its continuation.

These two translations admit different reasoning principles, however.

The double-negation translation supports encoding control effects but complicates compositional reasoning. A computation is free to duplicate its continuation and save it for later, or call a saved continuation, since the type of computations only requires that the computation returns $\bot$ and every continuation has the answer type $\bot$. However, it becomes difficult to give a compositional interpretation of programs compared to a standard λ-calculus. In ECC^D, for example, we define the meaning of expressions simply by evaluation to a value. Using the double-negation translation, CPS programs never return, so we cannot easily define the value of an expression by evaluation; we have to complete it first to get a whole program, then evaluate it, then look at the final state.

The locally polymorphic answer type translation makes it difficult to implement control effects, but easily supports compositional reasoning. Since each computation is parametric in its answer type, we are essentially forced to call the continuation exactly once and as the final step in the computation. If we want to encode control effects, this is a problem. However, it supports compositional reasoning. We can locally evaluate CPS'd expressions and get the meaning of that expression as a value by picking a meaningful answer type. For example, given the computation $e : \forall\alpha.(Bool \to \alpha) \to \alpha$, we can either treat it as a program and instantiate the answer type with $\bot$ ($e\ \bot : (Bool \to \bot) \to \bot$), or get the meaning of the underlying value by instantiating the answer type with $Bool$, the type of the underlying value, and applying the identity function as the continuation $e\ Bool\ \lambda x.x : Bool$. The expression $e\ Bool\ \lambda x.x$ will return the underlying value of e, supporting compositional reasoning.

The standard translation for type-preserving compilation is the double-negation translation (Morrisett et al., 1999), but extending this translation to dependent types has proven challenging. Barthe et al. (1999) showed how to scale typed call-by-name (CBN) CPS translation to a large class of Pure Type Systems (PTSs), including the Calculus of Constructions (CC) without Σ types. To avoid certain technical difficulties (which I discuss in Section 6.5), they consider only *domain-free* PTSs, a variant of PTSs where λ abstractions do not carry the domain of their bound variable—*i.e.*, they are of the form $\lambda x.\, e$ instead of $\lambda x : A.\, e$ as in the *domain-ful* variant. Barthe and Uustalu (2002) tried to extend these results to the Calculus of Inductive Constructions (CIC), but ended up reporting a negative result, namely that the CBN CPS double-negation translation is *not type preserving* for dependent pairs. They go on to prove a general impossibility result: for dependent conditionals (in particular, sum types with dependent case analysis) type preservation is *not possible* when the CPS translation admits unrestricted control effects.

In this chapter, I use the locally polymorphic CPS translation and prove type preservation for the core features of dependency that were proven "impossible" by Barthe and Uustalu. The key is that the polymorphic CPS uses parametricity to guarantee absence of control effects. This first step is an over-approximation; we only need to prevent control effects in the presence of certain dependencies. However, it allows us to ignore effects and focus only on dependent-type preservation.

6.2 Main Ideas

Intuitively, in a dependently typed language, the power of the type system comes from the ability to express decidable equality between terms and types. These equalities are decided by reducing expressions to canonical forms and checking that the resulting *values* are syntactically identical. In the source language—*i.e.*, before CPS translation—since the language is effect-free, every term can be thought of as a value since every expression reduces to a value. But CPS translation converts source expressions into computations of type $(A \to \bot) \to \bot$. This changes the *interface* to the values—now we can only access the value indirectly, by providing a computation that will do something with the value. In essence, ensuring CPS translations are type-preserving is hard because every source value has turned into a computation whose underlying value isn't directly accessible for purposes of deciding equivalence. In particular, with the double-negation translation, one cannot recover the underlying value, because every continuation must return $\bot$.

This description in terms of interfaces is just a shallow description of the problem. At a deeper level, the problem is that dependently typed languages rely on the ability of the type system to copy expressions from a *term-level* context into a *type-level* context, but CPS transforms expressions into *computations* whose meaning, or underlying value, depends on its term-level context. This copying happens in particular in the elimination rules for features related to dependency—*i.e.*, dependent functions, dependent pairs, and dependent conditionals—hence these features are at the heart of past negative results. After CPS, we no longer copy an expression, whose meaning is self-contained; instead we copy a computation, whose meaning depends on its term-level context. Not only do we "forget" part of the meaning of computations, but as we discussed before, a computation cannot run in a type-level context—it requires a term-level context. As I describe next, the solution to these problems will be to record part of the term-level contexts during type checking and to provide an interface that allows types to run computations.

To make this intuition concrete, I present two examples. I focus on two cases of the double-negation translation that fail to type check: the CBN translation of $\mathsf{snd}\,\mathsf{e}$ (reported by Barthe and Uustalu (2002)) and the CBV translation of $\mathsf{e}\ \mathsf{e}'$.

Consider the CBN CPS translation. We translate a term e of type A into a CPS'd computation, written $e^{\div}$, of type $A^{\div}$. Given a type A, we define its *computation translation* $A^{\div}$ and its *value translation* A^{+}. Below, I define the translations for dependent pairs and dependent functions. As in Chapter 5, the translations are defined by induction on typing derivations, but I present them less formally in this section.

$$A^{\div} \overset{\text{def}}{=} (A^{+} \rightarrow \bot) \rightarrow \bot \qquad (\Sigma\, x : A.\, B)^{+} \overset{\text{def}}{=} \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div} \qquad (\Pi\, x : A.\, B)^{+} \overset{\text{def}}{=} \boldsymbol{\Pi}\, \mathbf{x} : A^{\div}.\, B^{\div}$$

Note that since this is the CBN translation, the translated argument type for dependent functions is a computation type $A^{\div}$ instead of a value type A^{+}, and the translated component types for dependent pairs are computation types $A^{\div}$ and $B^{\div}$.

As a warm-up, consider the CBN translation of $\mathsf{fst}\, e$ (where $e : \Sigma\, x : A.\, B$):

$$\begin{aligned}(\mathsf{fst}\, e : A)^{\div} \overset{\text{def}}{=} {} & \boldsymbol{\lambda}\, \mathbf{k} : A^{+} \rightarrow \bot. \\ & e^{\div}\ (\boldsymbol{\lambda}\, \mathbf{y} : (\boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div}). \\ & \qquad \mathbf{let}\ \mathbf{z} = (\mathbf{fst}\ \mathbf{y}) : A^{\div}\ \mathbf{in}\ \mathbf{z}\ \mathbf{k})\end{aligned}$$

It is easy to see that the above type checks (checking the types of $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{k}$).

Next, consider the CBN translation of $\mathsf{snd}\, e$:

$$\begin{aligned}(\mathsf{snd}\, e : B[\mathsf{fst}\, e/x])^{\div} \overset{\text{def}}{=} {} & \boldsymbol{\lambda}\, \mathbf{k} : B^{+}[(\mathsf{fst}\, e)^{\div}/\mathbf{x}] \rightarrow \bot. \\ & e^{\div}\ (\boldsymbol{\lambda}\, \mathbf{y} : (\boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div}). \\ & \qquad \mathbf{let}\ \mathbf{z} = (\mathbf{snd}\ \mathbf{y}) : B^{\div}[\mathbf{fst}\ \mathbf{y}/\mathbf{x}]\ \mathbf{in}\ \mathbf{z}\ \mathbf{k})\end{aligned}$$

The above does not type check because the computation $\mathbf{z}$ expects a continuation of type $B^{+}[\mathbf{fst}\ \mathbf{y}/\mathbf{x}] \rightarrow \bot$ but $\mathbf{k}$ has type $B^{+}[(\mathsf{fst}\, e)^{\div}/\mathbf{x}] \rightarrow \bot$. Somehow we need to show that $\mathbf{fst}\ \mathbf{y} \equiv (\mathsf{fst}\, e)^{\div}$. But what is the relationship between $\mathbf{y}$ and e? Intuitively, $e^{\div} : A^{\div}$ is a computation that will pass its result—*i.e.*, the underlying value of type A^{+} inside $e^{\div}$, which corresponds to the value produced by evaluating the source term e—to its continuation. So when $e^{\div}$'s continuation is called, its argument $\mathbf{y}$ will always be equal to the unique underlying value inside $e^{\div}$. However, since we have used a *function* to describe a continuation, we must type check the body of the continuation assuming that $\mathbf{y}$ is *any* value of the appropriate type instead of the *exactly one* underlying value from $e^{\div}$.

Even if we could communicate that $\mathbf{y}$ is equal to exactly one value, we have no way to extract the underlying A^{+} value from $e^{\div}$ since the latter takes a continuation that never returns (since it must return a term of type $\bot$). To extract the underlying value from a computation, we need a means of converting from $A^{\div}$ to A^{+}. In essence, after CPS, we have an *interoperability* problem between the term language (where computations have type $A^{\div}$) and the type language (which needs values of type A^{+}). In the source language, before CPS, we are able to pretend that the term and type languages are the same because all computations of type A reduce to values of type A. However, the CPS translation creates a gap between the term and type languages; it changes the *interface* to terms so that the only way to get a value out of a computation is to have a continuation, which can never return, ready to receive that value.

The locally polymorphic answer type translation solves both of the above problems. We change the computation translation to $A^{\div} = \Pi\, \alpha : \star. (A^{+} \to \alpha) \to \alpha$. Now, to extract the underlying value of type A^{+} from $e^{\div} : A^{\div}$, we can run $e^{\div}$ with the identity continuation as follows: $e^{\div}\ A^{+}\ \mathbf{id}$. Moreover, we can now justify type checking the body of $e^{\div}$'s continuation under the assumption that $\mathbf{y} \equiv e^{\div}\ A^{+}\ \mathbf{id}$ thanks to a *free theorem* we get from the type $A^{\div}$. The free theorem says that running some $\mathbf{e} : A^{\div}$ with continuation $\mathbf{k} : \mathbf{A} \to \mathbf{B}$ is equivalent to running $\mathbf{e}$ with the identity continuation and then passing the result to $\mathbf{k}$, *i.e.*, $\mathbf{e}\ \mathbf{B}\ \mathbf{k} \equiv \mathbf{k}\ (\mathbf{e}\ \mathbf{A}\ \mathbf{id})$.

To formalize this intuition in the target language, I first add new syntax for the application of a computation to its answer type and continuation: $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$. Next, I internalize the aforementioned free theorem by adding two rules to the target language. The first is the following typing rule which records (a representation of) the value of a computation while type checking a continuation. That is, it allows us to assume $\mathbf{y} \equiv e^{\div}\ A^{+}\ \mathbf{id}$ when type checking the body of $e^{\div}$'s continuation.

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{e} : \Pi\, \alpha : \star. (\mathbf{A} \to \alpha) \to \alpha \\ \Gamma \vdash \mathbf{B} : \star \qquad \Gamma, \mathbf{x} = \mathbf{e}\ \mathbf{A}\ \mathbf{id} \vdash \mathbf{e}' : \mathbf{B}\end{array}}{\Gamma \vdash \mathbf{e}\ @\ \mathbf{B}\ (\lambda\, \mathbf{x} : \mathbf{A}. \mathbf{e}') : \mathbf{B}}\ \textsc{Rule T-Cont}$$

The second is the following equivalence rule, which is justified by the free theorem. Intuitively, this rule normalizes CPS'd computations to the "value" $e^{\div}\ A^{+}\ \mathbf{id}$.

$$\frac{}{\Gamma \vdash (\mathbf{e}_1\ @\ \mathbf{B}\ (\lambda\, \mathbf{x} : \mathbf{A}. \mathbf{e}_2)) \equiv (\lambda\, \mathbf{x} : \mathbf{A}. \mathbf{e}_2)\ (\mathbf{e}_1\ \mathbf{A}\ \mathbf{id})}\ \textsc{Rule} \equiv\textsc{-Cont}$$

I prove these rules admissable in Section 6.4.

Here is the updated CPS translation $(\mathsf{snd}\ \mathsf{e} : \mathsf{B}[\mathsf{fst}\ \mathsf{e}/\mathsf{x}])^{\div}$ that leverages answer-type polymorphism:

$$\begin{array}{l}\lambda\, \alpha : \star. \lambda\, \mathbf{k} : B^{+}[(\mathsf{fst}\ \mathsf{e})^{\div}/\mathbf{x}] \to \alpha. \\ \qquad\qquad e^{\div}\ @\ \alpha\ (\lambda\, \mathbf{y} : (\Sigma\, \mathbf{x} : A^{\div}. B^{\div}). \mathbf{let}\ \mathbf{z} = (\mathbf{snd}\ \mathbf{y}) : B^{\div}[\mathbf{fst}\ \mathbf{y}/\mathbf{x}]\ \mathbf{in}\ \mathbf{z}\ \alpha\ \mathbf{k})\end{array}$$

To type check $e^{\div}\ @\ \alpha\ \ldots$ we use Rule T-Cont. When type checking the body of $e^{\div}$'s continuation, we have that $\mathbf{y} \equiv e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}. B^{\div})\ \mathbf{id}$ and recall that we need to show that $\mathbf{fst}\ \mathbf{y} \equiv (\mathsf{fst}\ \mathsf{e})^{\div}$. This requires expanding $(\mathsf{fst}\ \mathsf{e})^{\div}$ and making use of the Rule ≡-Cont rule we now have available in the target language. Here is an informal sketch of the proof—I give the detailed proof in Section 6.5.1.

$$\begin{aligned}(\mathsf{fst}\ \mathsf{e})^{\div} &\equiv e^{\div}\ @\ \alpha'\ (\lambda\, \mathbf{y}. \mathbf{fst}\ \mathbf{y}) && \text{by (roughly) the translation} \quad (77)\\ &\equiv (\lambda\, \mathbf{y}. \mathbf{fst}\ \mathbf{y})\ (e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}. B^{\div})\ \mathbf{id}) && \text{by Rule} \equiv\textsc{-Cont} \quad (78)\\ &\equiv \mathbf{fst}\ (e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}. B^{\div})\ \mathbf{id}) && \text{by reduction} \quad (79)\\ &\equiv \mathbf{fst}\ \mathbf{y} && \text{since } \mathbf{y} \equiv e^{\div}\ (\Sigma\, \mathbf{x} : A^{\div}. B^{\div})\ \mathbf{id} \quad (80)\end{aligned}$$

Notice that the CPS translation—as well as the new rules Rule T-Cont and Rule ≡-Cont—only uses the new @ syntax for certain applications. Intuitively, we only need to use @ only when type checking requires the free theorem. This happens when CPS translating a depended-upon computation, such as for e in $\mathsf{snd}\,\mathsf{e}$.

Next, let's look at the translation of dependent function types. Again, we start with a warm-up; consider the following CBN double-negation translation of $\mathsf{e}\ \mathsf{e'}$ (where $\mathsf{e} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ and $\mathsf{e'} : \mathsf{A}$):

$$(\mathsf{e}\ \mathsf{e'} : \mathsf{B}[\mathsf{e'}/\mathsf{x}])^{\div} = \boldsymbol{\lambda}\,\mathbf{k} : (\mathsf{B}^{+}[\mathsf{e'}^{\div}/\mathbf{x}]) \to \bot.\,\mathsf{e}^{\div}\ (\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{\div}.\,\mathsf{B}^{\div}.\,(\mathbf{f}\ \mathsf{e'}^{\div})\ \mathbf{k})$$

The above type checks (as seen by inspecting the types of $\mathbf{f}$ and $\mathbf{k}$). Notice that $\mathsf{e'}^{\div}$ appears as an argument to $\mathbf{f}$ so the type of $\mathbf{f}\ \mathsf{e'}^{\div} : \mathsf{B}^{\div}[\mathsf{e'}^{\div}/\mathbf{x}]$.

Now consider the CBV CPS translation based on *double negation*, which fails to type check. We define the CBV computation translation $\mathsf{A}^{\div}$ and value translation A^{+} as follows.

$$\mathsf{A}^{\div} = (\mathsf{A}^{+} \to \bot) \to \bot \qquad (\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B})^{+} = \boldsymbol{\Sigma}\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{B}^{+} \qquad (\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B})^{+} = \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{B}^{\div}$$

Since this is a CBV translation, the translated argument for the dependent function is a value of type A^{+} and the translated component types for dependent pairs are values of types A^{+} and B^{+}.

Here is the CBV CPS translation $\mathsf{e}\ \mathsf{e'}$ (where $\mathsf{e} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ and $\mathsf{e'} : \mathsf{A}$):

$$(\mathsf{e}\ \mathsf{e'} : \mathsf{B}[\mathsf{e'}/\mathsf{x}])^{\div} = \boldsymbol{\lambda}\,\mathbf{k} : (\mathsf{B}^{+}[\mathsf{e'}^{+}/\mathbf{x}]) \to \bot.\,\mathsf{e}^{\div}\ (\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{B}^{\div}.\,\mathsf{e'}^{\div}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathsf{A}^{+}.\,(\mathbf{f}\ \mathbf{x})\ \mathbf{k}))$$

For the moment, ignore that our type annotation on $\mathbf{k}$, $(\mathsf{B}^{+}[\mathsf{e'}^{+}/\mathbf{x}])$, seems to require a value translation of terms $\mathsf{e'}^{+}$, which we can't normally define. Instead, notice that unlike in the CBN translation, we now evaluate the argument $\mathsf{e'}^{\div}$ before calling $\mathbf{f}$, so in CBV we have the application $\mathbf{f}\ \mathbf{x} : \mathsf{B}^{\div}[\mathbf{x}/\mathbf{x}]$. This translation fails to type check since the computation $\mathbf{f}\ \mathbf{x}$ expects a continuation of type $(\mathsf{B}^{+}[\mathbf{x}/\mathbf{x}]) \to \bot$ but $\mathbf{k}$ has type $(\mathsf{B}^{+}[\mathsf{e'}^{+}/\mathbf{x}]) \to \bot$. Somehow we need to show that $\mathbf{x} \equiv \mathsf{e'}^{+}$. This situation is almost identical to what we saw with the failing CBN translation of $\mathsf{snd}\,\mathsf{e}$. Analogously, this time we ask what is the relationship between $\mathbf{x}$ and $\mathsf{e'}^{\div}$, or $\mathsf{e'}^{+}$? As before, note that the only value that can flow into $\mathbf{x}$ is the unique underlying value in $\mathsf{e'}^{\div}$.

Hence, fortunately, the solution is again to do what we did for the CBN translation: adopt a CPS translation based on answer-type polymorphism. As before, we change the computation translation to $\mathsf{A}^{\div} = \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(\mathsf{A}^{+} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$. Here is the updated CBV CPS translation of $(\mathsf{e}\ \mathsf{e'} : \mathsf{B}[\mathsf{e'}/\mathsf{x}])^{\div}$:

$$\begin{aligned}&\boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\,\boldsymbol{\lambda}\,\mathbf{k} : (\mathsf{B}^{+}[(\mathsf{e'}^{\div}\ \mathsf{A}^{+}\ \mathbf{id})/\mathbf{x}]) \to \boldsymbol{\alpha}.\\ &\quad \mathsf{e}^{\div}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{B}^{\div}.\\ &\qquad \mathsf{e'}^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathsf{A}^{+}.\,(\mathbf{f}\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k}))\end{aligned}$$

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

Figure 6.1: CoC^D Syntax

First, notice that this uses the new **@** form when evaluating the argument $\mathsf{e}'^{\div}$, which tells us we're using our new typing rule to record the value of $\mathsf{e}'^{\div}$ while we type check its continuation. Second, notice the type annotation on $\mathbf{k}$. Earlier I observed that the type annotation for $\mathbf{k}$, $(\mathsf{B}^{+}[\mathsf{e}'^{+}/\mathbf{x}])$, seemed to require a value translation on terms e'^{+} that cannot normally be defined. The translation gives us a sensible way of modeling the value translation of a term by invoking a computation with the identity continuation—so e'^{+} is just the underlying value in $\mathsf{e}'^{\div}$, *i.e.*, $(\mathsf{e}'^{\div}\ \mathsf{A}^{+}\ \mathbf{id})$. This is an important point to note: unlike CBN CPS, where we can substitute computations for variables, in CBV CPS we must find a way to extract the underlying value from computations of type $\mathsf{A}^{\div}$ since variables expect values of type A^{+}. Without answer-type polymorphism, CBV CPS is, in some sense, much more broken than CBN CPS! Indeed, Barthe et al. (1999) already gave a CBN double-negation translation for dependent functions, but typed CBV double-negation translation for dependent function fails.

Using the new typing rule and equivalence rules from earlier, we are able to type check the above translation of $\mathsf{e}\ \mathsf{e}'$ in essentially the same way as for the CBN translation of $\mathsf{snd}\ \mathsf{e}$. I show the detailed proof in Section 6.6.1.

The reader may worry that our CBV CPS translation produces many terms of the form $\mathbf{k}\ (\mathsf{e}^{\div}\ \mathsf{A}^{+}\ \mathbf{id})$, which aren't really in CPS since $\mathsf{e}^{\div}\ \mathsf{A}^{+}\ \mathbf{id}$ must return. However, notice that these only appear in types, not terms. That is, we only run a computation with the identity continuation to convert a CPS computation into a value *in the types* for deciding equivalence. The run-time terms are all in CPS and can be run in a machine-like semantics in which computations never return.

6.3 The Calculus of Constructions with Definitions

The language CoC^D is an extension of the Calculus of Constructions (*CoC*) with booleans, dependent pairs and definitions. This is a restriction of the earlier ECC^D, without dependent conditionals, higher universes, and cumulativity. Note that the booleans in CoC^D do not allow dependent elimination; they only serve as a ground type for observations across languages. I return to dependent conditions in Section 6.7. I

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

$$\begin{array}{rcll}
\Gamma \vdash (\lambda\, x : A.\, e_1)\ e_2 & \rhd_\beta & e_1[e_2/x] & \\
\Gamma \vdash \mathsf{fst}\ \langle e_1, e_2 \rangle & \rhd_{\pi_1} & e_1 & \\
\Gamma \vdash \mathsf{snd}\ \langle e_1, e_2 \rangle & \rhd_{\pi_2} & e_2 & \\
\Gamma \vdash \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_1} & e_1 & \\
\Gamma \vdash \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_1} & e_2 & \\
\Gamma \vdash x & \rhd_\delta & e & \text{where } x = e : A \in \Gamma \\
\Gamma \vdash \mathsf{let}\ x = e_2 : A\ \mathsf{in}\ e_1 & \rhd_\zeta & e_1[e_2/x] &
\end{array}$$

Figure 6.2: CoC^D Reduction

adapt this presentation from the model of the Calculus of Inductive Constructions (*CIC*) given in the Coq reference manual (The Coq Development Team, 2017, Chapter 4).

I present the syntax of CoC^D in Figure 6.1 in the style of a Pure Type System (*PTS*) with no syntactic distinction between terms and types and *kinds*, which describe types. This has been true of earlier chapters, but we will soon make an explicit distinction in order to selectively CPS translate only terms, as, unlike with ANF and closure conversion, it is unclear how to uniformly CPS translate types and kinds in addition to terms. As before, I use the phrase expression to refer to a term, type, or kind in the PTS syntax. I usually use the meta-variable e to evoke a term expression and I use t for an expression to be explicitly ambiguous about its nature as a term, type, or kind.

The language includes one impredicative universe, or *sort*, $\star$, and its type, $\square$. Compared to ECC^D from Chapter 2, we can think of $\star$ as Prop and $\square$ as Type_1. The syntax of expressions includes the universe $\star$, variables x or α, dependent function types $\Pi\, x : A.\, B$, dependent functions $\lambda\, x : A.\, e$, application $e_1\ e_2$, dependent let $\mathsf{let}\ x = e : A\ \mathsf{in}\ e'$, dependent pair types $\Sigma\, x : A.\, B$, dependent pairs $\langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B$, and first and second projections $\mathsf{fst}\, e$ and $\mathsf{snd}\, e$. Note that, unlike in ECC^D, we cannot write $\square$ in source programs—it is only used by the type system. The typing environment Γ includes assumptions $x : A$ and definitions $x = e : A$. Note that definitions in this language include annotations, unlike in ECC^D. Unlike in the ANF translation in Chapter 4, including annotations on definitions does not complicate the CPS translation because the CPS translation is already inherently complicated by producing type annotations for continuations (which I discuss more later).

As with dependent pairs, I omit the type annotations on let expressions, $\mathsf{let}\ x = e\ \mathsf{in}\ e'$, when they are clear from context. I use the notation $A \to B$ for a function type whose result B does not depend on the input.

The reduction relation, conversion relation, and equivalence relations for CoC^D are essentially the same as for ECC^D. I partially duplicate them here anyway for convenience, and give full definitions in Appendix F. Notice, however, that there is no subtyping relation since CoC^D excludes higher universes. In Figure 6.2, I present the reduction

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

$$\frac{\Gamma \vdash e \rhd^* e_1 \qquad \Gamma \vdash e' \rhd^* e_1}{\Gamma \vdash e \equiv e'}\ \equiv$$

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

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

Figure 6.3: CoC^D Equivalence

relation for CoC^D, and in Figure 6.3, I present the equivalence relation. Note that CoC^D does not fix an evaluation order, but this is not important since CoC^D is effect-free.

The typing rules for CoC^D, Figure 6.4, are also essentially the same, but missing removed features. The judgment $\vdash \Gamma$ checks that the environment Γ is well-formed; it is defined by mutual recursion with the typing judgment. Note that there is no typing rule analogous to Rule TYPE, so $\square$ is not a well-typed expression. This means $\square$ does not need a type, and means CoC^D excludes higher universes. Since I exclude higher universes, I exclude cumulativity, so the Rule CONV is different. Notice that the Rule IF does not allow the result type to be dependent; I return to this in Section 6.7. Instead of subtyping, Rule CONV allows typing an expression $e : A$ as $e : B$ when $A \equiv B$. The remaining rules are essentially the same. Rule PI-* essentially corresponds to Rule PI-PROP, and implicitly allows impredicativity in $\star$, since the domain A could be in the higher universe $\square$. Rule PI-$\square$ is predicative, similar to Rule PI-TYPE.

Note that for simplicity, I include only term-level pairs of type $\Sigma\, x : A.\, B : \star$. Type-level pairs $\Sigma\, x : A.\, B : \square$ introduce numerous minor difficulties with CPS translation. For instance, we can write pairs of terms and types $\langle e, A\rangle$ or $\langle A, e\rangle$. It is unclear how these expression should be CPS translated; should they be considered terms or types? This is an instance of the more general problem of computational relevance. In general, dependent-type-preserving compilation is difficult when we try to compile only the computationally relevant terms, because we cannot easily decide relevance. I discuss this further in Chapter 8.

To make the upcoming CPS translation easier to follow, I present a second version of the syntax for CoC^D in which we make the distinction between terms, types, and kinds explicit (see Figure 6.5). The two presentations are equivalent (Barthe et al., 1999), at least as long as we do not include computationally relevant higher universes. Distinguishing terms from types and kinds is useful since we only want to CPS translate *terms*, because our goal is to internalize only *run-time* evaluation contexts. I discuss *pervasive translation*, which also internalizes the type-level evaluation context, in Section 6.7.

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

$$\frac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \star : \square}\ * \qquad \frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi x : A.\, B : \star}\ \textsc{Pi-}*$$

$$\frac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi x : A.\, B : \square}\ \textsc{Pi-}\square \qquad \frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash \Pi x : A.\, B : U}{\Gamma \vdash \lambda x : A.\, e : \Pi x : A.\, B}\ \textsc{Lam}$$

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

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

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : B \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B}\ \textsc{If}$$

$$\frac{\Gamma \vdash e' : A \qquad \Gamma, x = e' : A \vdash e : B}{\Gamma \vdash \mathsf{let}\, x = e' : A\, \mathsf{in}\, e : B[e'/x]}\ \textsc{Let} \qquad \frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : B}\ \textsc{Conv}$$

Figure 6.4: CoC^D Typing

6.4 CPS Intermediate Language

The target language CoC^k is CoC^D extended with a syntax for parametric reasoning about computations in CPS, as discussed in Section 6.2. I present these extensions formally in Figure 6.6, and give the complete language definition in Appendix G. I add the form $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e'}$ to the syntax of CoC^k. This form represents a computation $\mathbf{e}$ applied to the answer type $\mathbf{A}$ and the continuation $\mathbf{e'}$. The reduction semantics is the same as that of the standard application. The equivalence rule Rule $\equiv$-Cont states that a computation $\mathbf{e}$ applied to its continuation $\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\, \mathbf{e'}$ is equivalent to the application of that continuation to the underlying value of $\mathbf{e}$. We extract the underlying value by applying $\mathbf{e}$ to the *halt continuation*, encoded as the identity function in this system. Rule T-Cont is used to type check applications that use the new $@$ syntax. This typing rule internalizes the fact that a continuation will be applied to one particular input, rather than an arbitrary value. It tells the type system that the application of a computation to a continuation $\mathbf{e}\ @\ \mathbf{A}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\, \mathbf{e'})$ jumps to the continuation $\mathbf{e'}$ after evaluating $\mathbf{e}$ to a value and binding the result to $\mathbf{x}$. We check the body of the continuation $\mathbf{e'}$ under the assumption that $\mathbf{x} = \mathbf{e}\ \mathbf{B}\ \mathbf{id}$, *i.e.*, with the equality that the

$$
\begin{array}{llcl}
\textit{Kinds} & \mathsf{K} & ::= & \star \mid \Pi\,\alpha : \mathsf{K}.\,\mathsf{K} \mid \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{K} \\
\textit{Types} & \mathsf{A}, \mathsf{B} & ::= & \alpha \mid \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \mid \Pi\,\alpha : \mathsf{K}.\,\mathsf{B} \mid \lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \mid \lambda\,\alpha : \mathsf{K}.\,\mathsf{B} \mid \mathsf{A}\ \mathsf{e} \\
 & & \mid & \mathsf{A}\ \mathsf{B} \mid \Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \mid \mathsf{bool} \mid \mathsf{let}\,\alpha = \mathsf{A} : \mathsf{K}\,\mathsf{in}\,\mathsf{B} \\
 & & \mid & \mathsf{let}\,\mathsf{x} = \mathsf{e} : \mathsf{A}\,\mathsf{in}\,\mathsf{B} \\
\textit{Terms} & \mathsf{e} & ::= & \mathsf{x} \mid \lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e} \mid \lambda\,\alpha : \mathsf{K}.\,\mathsf{e} \mid \mathsf{e}\ \mathsf{e} \mid \mathsf{e}\ \mathsf{A} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle\,\mathsf{as}\,\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} \\
 & & \mid & \mathsf{fst}\,\mathsf{e} \mid \mathsf{snd}\,\mathsf{e} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\,\mathsf{e}\,\mathsf{then}\,\mathsf{e}_1\,\mathsf{else}\,\mathsf{e}_2 \\
 & & \mid & \mathsf{let}\,\mathsf{x} = \mathsf{e} : \mathsf{A}\,\mathsf{in}\,\mathsf{e} \mid \mathsf{let}\,\alpha = \mathsf{A} : \mathsf{K}\,\mathsf{in}\,\mathsf{e} \\
\textit{Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x} : \mathsf{A} \mid \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A}, \mid \Gamma, \alpha : \mathsf{K} \mid \Gamma, \alpha = \mathsf{A} : \mathsf{K}
\end{array}
$$

Figure 6.5: CoC^D Explicit Syntax

name $\mathbf{x}$ refers to the underlying value in the computation $\mathbf{e}$, which we access using the interface given by the polymorphic answer type.

Rule $\equiv$-CONT is a declarative rule that requires explicit symmetry and transitivity rules to complete the definition. I conjecture that the algorithmic versions look something like the following.

$$\frac{\Gamma \vdash \mathbf{e} \triangleright^* (\mathbf{e}_1\ @\ \mathbf{A}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{e}_2)) \qquad \Gamma \vdash (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{e}_2)\ (\mathbf{e}_1\ \mathbf{B}\ \mathbf{id}) \equiv \mathbf{e}'}{\Gamma \vdash \mathbf{e} \equiv \mathbf{e}'}\ \equiv\text{-CONT}_1$$

$$\frac{\Gamma \vdash \mathbf{e}' \triangleright^* (\mathbf{e}_1\ @\ \mathbf{A}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{e}_2)) \qquad \Gamma \vdash \mathbf{e} \equiv (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{e}_2)\ (\mathbf{e}_1\ \mathbf{B}\ \mathbf{id})}{\Gamma \vdash \mathbf{e} \equiv \mathbf{e}'}\ \equiv\text{-CONT}_2$$

Note that Rule $\equiv$-CONT and Rule T-CONT internalize a specific "free theorem" that we need in order to prove type preservation of the CPS translation. In particular, Rule $\equiv$-CONT only holds when the CPS'd computation $\mathbf{e}_1$ has the expected parametric type $\boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(\mathbf{A} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$ given in Rule T-CONT. Notice, however, that our statement of Rule $\equiv$-CONT does not put any requirements on the type of $\mathbf{e}_1$. This is because we use an untyped equivalence, and this untyped equivalence is necessary to prove type preservation (see Section 6.5.1). Therefore, we cannot simply add typing assumptions directly to Rule $\equiv$-CONT. Instead, we must rely on the fact that the term $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ has only one introduction rule, Rule T-CONT. Since there is only one applicable typing rule, anytime $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ appears in our type system, $\mathbf{e}$ has the required parametric type. Furthermore, while equivalence is untyped, we never appeal to equivalence with ill-typed terms; we only refer to the equivalence $\mathbf{A}' \equiv \mathbf{B}'$ in Rule CONV *after* checking that both $\mathbf{A}'$ and $\mathbf{B}'$ are well-typed. For example, suppose the term $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ occurs in type $\mathbf{A}'$, and to prove that $\mathbf{A}' \equiv \mathbf{B}'$ requires our new rule Rule $\equiv$-CONT. Because $\mathbf{A}'$ is well-typed, we know that its subterms, including $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$, are well-typed. Since $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{e}'$ can only be well-typed by Rule T-CONT, we know $\mathbf{e}$ has the required parametric type.

Extensions to Syntax, Figure 6.1

$$\textit{Terms} \quad \mathbf{e} \quad ::= \quad \cdots \mid \mathbf{e} \mathbin{@} \mathbf{A}\ \mathbf{e}'$$

Extensions to Reduction and Equivalence, Figure F.4

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

$$\lambda\, \alpha : \star.\, \mathbf{e}_1 \mathbin{@} \mathbf{A}\ \mathbf{e}_2 \quad \triangleright_{@} \quad (\mathbf{e}_1[\mathbf{A}/\alpha])\ \mathbf{e}_2$$
$$\vdots$$

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

$$\cdots \qquad \frac{}{\Gamma \vdash (\mathbf{e}_1 \mathbin{@} \mathbf{A}\ (\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}_2)) \equiv (\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}_2)\ (\mathbf{e}_1\ \mathbf{B}\ \mathbf{id})}\ \equiv\text{-Cont}$$

Extensions to Typing, Figure 6.4

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

$$\cdots$$

$$\frac{\Gamma \vdash \mathbf{e} : \Pi\, \alpha : \star.\, (\mathbf{B} \to \alpha) \to \alpha \qquad \Gamma \vdash \mathbf{A} : \star \qquad \Gamma, \mathbf{x} = \mathbf{e}\ \mathbf{B}\ \mathbf{id} \vdash \mathbf{e}' : \mathbf{A}}{\Gamma \vdash \mathbf{e} \mathbin{@} \mathbf{A}\ (\lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}') : \mathbf{A}}\ \text{T-Cont}$$

Figure 6.6: CoC^k: CoC^D with CPS Extensions (excerpts)

Finally, notice that in Rule T-Cont and Rule ≡-Cont I use standard application syntax for the term $\mathbf{e}\ \mathbf{B}\ \mathbf{id}$. The new $@$ syntax is only necessary in the CPS translation when we require one of our new rules to type check the translated term. The type of the identity function doesn't depend on any value, so we never need Rule T-Cont to type check the identity continuation. In a sense, $\mathbf{e}\ \mathbf{B}\ \mathbf{id}$ is the normal form of a CPS'd "value" so we never need Rule ≡-Cont to rewrite this term—*i.e.*, using Rule ≡-Cont to rewrite $\mathbf{e}\ \mathbf{B}\ \mathbf{id}$ to $\mathbf{id}\ (\mathbf{e}\ \mathbf{B}\ \mathbf{id})$ would just evaluate to the original term.

6.4.1 Meta-Theory

I prove that CoC^k is consistent by giving a model of CoC^k in the extensional CoC, following the standard architecture described in Chapter 3.

Typographical Note. *In this section, I write terms in extensional CoC using a italic, black, serif font.*

The idea behind the model is that we can translate each use of Rule ≡-Cont in CoC^k to a propositional equivalence in extensional CoC. Next, we translate any term

$\boxed{\Gamma \vdash e_1 \equiv e_2}$

All other rules identical to CoC^D

$$\frac{\Gamma \vdash p : e_1 = e_2}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-Ext}$$

Figure 6.7: Extensional CoC Equivalence

that is typed by Rule T-Cont into a dependent let. Finally, as before, we establish that if there were a proof of $\bot$ in CoC^k, our translation would construct a proof of $\bot$ in extensional CoC. But since extensional CoC is consistent, there can be no proof of $\bot$ in CoC^k.

As our model is in the extensional CoC, it is not clear that type checking in CoC^k is decidable. I believe that type checking should be decidable for all programs produced by the CPS translations, since type checking in the source language CoC^D is decidable. In the worst case, to ensure decidability we could change the translation to use a propositional version of Rule $\equiv$-Cont. The definitional presentation is simpler, but it should be possible to change the translation so that, in any term that currently relies on Rule $\equiv$-Cont, we insert type annotations that compute type equivalence using a propositional version of Rule $\equiv$-Cont. I leave the issue of decidability of type checking in CoC^k for future work.

Modeling [$\equiv$-Cont]

The extensional CoC differs from the source language CoC^D in one key way: it allows using the existence of a propositional equivalence as a definitional equivalence, as shown in Figure 6.7. The syntax and typing rules are otherwise similar to CoC^D presented in Section 6.3, but must be extended to include the identity type.

In extensional CoC, we can model each use of the definitional equivalence Rule $\equiv$-Cont by Rule $\equiv$-Ext, as long as there exists a proof $p : (e\ A\ k) = (k\ (e\ B\ id))$, *i.e.*, a propositional proof of Rule $\equiv$-Cont; I prove this propositional proof always exists by using the parametricity translation of Keller and Lasson (2012). This translation gives a parametric model of CoC in itself. This translation is based on prior translations that apply to all Pure Type Systems (Bernardy et al., 2012), but includes an impredicative universe and provides a Coq implementation that I use for the key proof.

The parametricity translation of a type A, written $[\![A]\!]^{\mathcal{P}}$, essentially transforms the type into a relation on terms of that type. On terms e of type A, the translation $[\![e]\!]^{\mathcal{P}}$ produces a proof that e is related to itself in the relation given by $[\![A]\!]^{\mathcal{P}}$. For example, a type $\star$ is translated to the relation $[\![\star]\!]^{\mathcal{P}} = \lambda\,(x, x' : \star).\, x \to x' \to \star$. The translation of a polymorphic function type $[\![\Pi\,\alpha : \star.\, A]\!]^{\mathcal{P}}$ is the following.

$$\lambda\,(f, f' : (\Pi\,\alpha : \star.\, A)).\, \Pi\,(\alpha, \alpha' : \star).\, \Pi\,\alpha_r : [\![\star]\!]^{\mathcal{P}}\ \alpha\ \alpha'.\, ([\![A]\!]^{\mathcal{P}}\ (f\ \alpha)\ (f'\ \alpha'))$$

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

All other rules are homomorphic

$$\dfrac{\mathbf{\Gamma} \vdash \mathbf{B} : \_ \rightsquigarrow_\circ B \qquad \mathbf{\Gamma} \vdash \mathbf{A} : \_ \rightsquigarrow_\circ A \qquad \begin{array}{c}\mathbf{\Gamma} \vdash \mathbf{e} : \_ \rightsquigarrow_\circ e \\ \mathbf{\Gamma}, \mathbf{x} = \mathbf{e}\ \mathbf{B}\ \mathbf{id} \vdash \mathbf{e'} : \mathbf{A} \rightsquigarrow_\circ e'\end{array}}{\mathbf{\Gamma} \vdash \mathbf{e}\ @\ \mathbf{A}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{e'}) : \mathbf{A} \rightsquigarrow_\circ \mathit{let}\, x = e\ B\ \mathit{id} : B\, \mathit{in}\, e'}\ \textsc{Un-Cont}$$

Figure 6.8: Model of CoC^k in Extensional CoC

This relation produces a proof that the bodies of functions f and f' are related when provided a relation α_r for the two types of α and α'. This captures the idea that functions at this type must behave parametrically in the abstract type α. This translation gives us Theorem 6.4.1 (Parametricity for extensional CoC), *i.e.*, that every expression in extensional CoC is related to itself in the relation given by its type.

Theorem 6.4.1 (Parametricity for extensional CoC). *If* $\Gamma \vdash t : t'$ *then* $[\![\Gamma]\!]^{\mathcal{P}} \vdash [\![t]\!]^{\mathcal{P}} : [\![t']\!]^{\mathcal{P}}\ t\ t$

I apply Theorem 6.4.1 to the CPS type $\Pi\,\alpha : \star.\,(B \to \alpha) \to \alpha$ to prove Lemma 6.4.2. Since a CPS'd term is a polymorphic function, we get to provide a relation α_r for the type α. The translation then gives us a proof that $e\ A\ k$ and $e\ B\ id$ are related by α_r, so we simply choose α_r to be a relation that guarantees $e\ A\ k = k\ (e\ B\ id)$. I formalize part of the proof in Coq, given in Appendix G. By Rule $\equiv$-Ext, Theorem 6.4.1, and the relation just described, we arrive at a proof of Lemma 6.4.2 for CPS'd computations encoded in the extensional CoC.

Lemma 6.4.2 (Continuation Shuffling). *If* $\Gamma \vdash A : \star$, $\Gamma \vdash B : \star$, $\Gamma \vdash k : B \to A$, *and* $\Gamma \vdash e : \Pi\,\alpha : \star.\,(B \to \alpha) \to \alpha$ *then* $\Gamma \vdash e\ A\ k \equiv k\ (e\ B\ id)$

Note that this lemma relies on the type of the term e. We must only appeal to this lemma, and the equivalence justified by it, when e has the right type. In CoC^k, this is guaranteed by the typing rule Rule T-Cont, as discussed earlier in this section.

Modeling T-Cont

In Figure 6.8, I present the key translation rule for modeling CoC^k in extensional CoC. All other rules are inductive on the structure of typing derivations. Note that since we only need to justify the additional typing rule Rule T-Cont, this is the only rule that is changed by the translation.

For brevity in the proofs, I define the following notation for the translation of expressions and from CoC^k into extensional CoC.

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

By writing $\mathbf{e}^{\circ}$, I refer to the expression produced by the translation with the typing derivation $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ as an implicit parameter.

First, I show that the definition of $\perp$ is preserved. As in Chapter 5, I define $\perp$ as $\mathbf{\Pi}\, \boldsymbol{\alpha} : \star.\, \boldsymbol{\alpha}$, *i.e.*, the function that accepts any proposition and returns a proof that the proposition holds. It is simple to see that this type has type $\star$ in CoC^{k} by the rule Rule Pi-*. Note that $\mathbf{\Pi}\, \boldsymbol{\alpha} : \star.\, \boldsymbol{\alpha}$ is translated to $\Pi\, \alpha : \star.\, \alpha$ of type $\star$, *i.e.*, $\perp$ is translated to $\perp$.

Lemma 6.4.3 (Preservation of Falseness). $\mathbf{\Gamma} \vdash (\mathbf{\Pi}\, \boldsymbol{\alpha} : \star.\, \boldsymbol{\alpha}) : \star \rightsquigarrow_{\circ} \Pi\, \alpha : \star.\, \alpha$

Again, I start by proving compositionality, a crucial lemma to both equivalence preservation and type preservation. The proof is straightforward by induction on the typing derivation of $\mathbf{e}$.

Lemma 6.4.4 (Compositionality). $(\mathbf{e}[\mathbf{e}'/\mathbf{x}])^{\circ} \equiv \mathbf{e}^{\circ}[\mathbf{e}'^{\circ}/x]$

Proof. By induction on the typing derivation of $\mathbf{e}$. There is one interesting case.

Case: Rule T-Cont $\mathbf{e} = \mathbf{e}_1\ @\ \mathbf{B}\ (\boldsymbol{\lambda}\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2)$ and $\mathbf{e}^{\circ} = \mathit{let}\, x' = (\mathbf{e}_1{}^{\circ}\ \mathbf{A}^{\circ}\ \mathit{id})\ \mathit{in}\, \mathbf{e}_2{}^{\circ}$

Without loss of generality, assume $\mathbf{x} \neq \mathbf{x}'$.

It suffices to show that

$$(\mathbf{e}_1[\mathbf{e}'/\mathbf{x}]\ @\ \mathbf{B}[\mathbf{e}'/\mathbf{x}]\ (\boldsymbol{\lambda}\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2)[\mathbf{e}'/\mathbf{x}])^{\circ} = (\mathit{let}\, x' = (\mathbf{e}_1{}^{\circ}\ \mathbf{A}^{\circ}\ \mathit{id})\ \mathit{in}\, \mathbf{e}_2{}^{\circ})[\mathbf{e}'^{\circ}/x]$$

$$\begin{aligned}
& (\mathbf{e}_1[\mathbf{e}'/\mathbf{x}]\ @\ \mathbf{B}[\mathbf{e}'/\mathbf{x}]\ (\boldsymbol{\lambda}\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2)[\mathbf{e}'/\mathbf{x}])^{\circ} \\
= \ & (\mathbf{e}_1[\mathbf{e}'/\mathbf{x}]\ @\ \mathbf{B}[\mathbf{e}'/\mathbf{x}]\ (\boldsymbol{\lambda}\, \mathbf{x}' : \mathbf{A}[\mathbf{e}'/\mathbf{x}].\, \mathbf{e}_2[\mathbf{e}'/\mathbf{x}]))^{\circ} && (81) \\
& \text{by definition of substitution} \\
= \ & (\mathit{let}\, x' = ((\mathbf{e}_1[\mathbf{e}'/\mathbf{x}])^{\circ}\ (\mathbf{A}[\mathbf{e}'/\mathbf{x}])^{\circ}\ \mathit{id})\ \mathit{in}\, (\mathbf{e}_2[\mathbf{e}'/\mathbf{x}])^{\circ}) && (82) \\
& \text{by definition of the translation} \\
= \ & (\mathit{let}\, x' = (\mathbf{e}_1{}^{\circ}[\mathbf{e}'^{\circ}/x]\ \mathbf{A}^{\circ}[\mathbf{e}'^{\circ}/x]\ \mathit{id})\ \mathit{in}\, \mathbf{e}_2{}^{\circ}[\mathbf{e}'^{\circ}/x]) && (83) \\
& \text{by the induction hypothesis} \\
= \ & (\mathit{let}\, x' = \mathbf{e}_1{}^{\circ}\ \mathbf{A}^{\circ}\ \mathit{id}\ \mathit{in}\, \mathbf{e}_2{}^{\circ})[\mathbf{e}'^{\circ}/x] && (84) \\
& \text{by definition of substitution}
\end{aligned}$$

□

The equivalence rules of extensional CoC with the addition of Lemma 6.4.2 (Continuation Shuffling) subsume the equivalence rules of CoC^{k}. Therefore, to show that equivalence is preserved, it suffices to show that conversion is preserved. I first show that reduction is preserved, Lemma 6.4.5, which easily implies preservation of conversion Lemma 6.4.6.

Lemma 6.4.5 (Preservation of Reduction). *If* $\mathbf{e}_1 \triangleright \mathbf{e}_2$*, then* $\mathbf{e}_1^{\circ} \triangleright^{*} e'$ *and* $\mathbf{e}_2^{\circ} \equiv e'$

Proof. By cases on the reduction step $\mathbf{e}_1 \triangleright \mathbf{e}_2$. There is one interesting case.

Case: $\mathbf{e} = (\lambda\, \alpha : \star.\, \mathbf{e}_1)\ @\ \mathbf{B}\ (\lambda\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2) \triangleright_{@} (\mathbf{e}_1[\mathbf{B}/\alpha])\ (\lambda\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2)$

By definition

$$\mathbf{e}^{\circ} = (let\, x' = ((\lambda\, \alpha : \star.\, \mathbf{e}_1)^{\circ}\ \mathbf{A}^{\circ}\ id)\, in\, \mathbf{e}_2{}^{\circ}) \triangleright_{\zeta} \mathbf{e}_2{}^{\circ}[((\lambda\, \alpha : \star.\, \mathbf{e}_1)^{\circ}\ \mathbf{A}^{\circ}\ id)/x']$$

We must show $((\mathbf{e}_1[\mathbf{B}/\alpha])\ (\lambda\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2))^{\circ} \equiv \mathbf{e}_2{}^{\circ}[((\lambda\, \alpha : \star.\, \mathbf{e}_1)^{\circ}\ \mathbf{A}^{\circ}\ id)/x']$.

$$\begin{aligned}
& ((\mathbf{e}_1[\mathbf{B}/\alpha])\ (\lambda\, \mathbf{x}' : \mathbf{A}.\, \mathbf{e}_2))^{\circ} \\
\equiv\ & ((\mathbf{e}_1{}^{\circ}[\mathbf{B}^{\circ}/\alpha])\ (\lambda\, x' : \mathbf{A}^{\circ}.\, \mathbf{e}_2{}^{\circ})) && (85) \\
& \text{by Lemma 6.4.4 and definition of } {}^{\circ} \\
\equiv\ & (\lambda\, \alpha : \star.\, \mathbf{e}_1{}^{\circ})\ \mathbf{B}^{\circ}\ (\lambda\, x' : \mathbf{A}^{\circ}.\, \mathbf{e}_2{}^{\circ}) && (86) \\
& \text{by Rule } \equiv \text{ and } \triangleright_{\beta} \\
\equiv\ & (\lambda\, x' : \mathbf{A}^{\circ}.\, \mathbf{e}_2{}^{\circ})\ ((\lambda\, \alpha : \star.\, \mathbf{e}_1{}^{\circ})\ \mathbf{A}^{\circ}\ id) && (87) \\
& \text{by Lemma 6.4.2 (Continuation Shuffling)} \\
\equiv\ & \mathbf{e}_2{}^{\circ}[((\lambda\, \alpha : \star.\, \mathbf{e}_1{}^{\circ})\ \mathbf{A}^{\circ}\ id)/x'] && (88) \\
& \text{by Rule } \equiv \text{ and } \triangleright_{\beta} \\
\equiv\ & \mathbf{e}_2{}^{\circ}[((\lambda\, \alpha : \star.\, \mathbf{e}_1)^{\circ}\ \mathbf{A}^{\circ}\ id)/x'] && (89) \\
& \text{by Lemma 6.4.4}
\end{aligned}$$

□

Lemma 6.4.6 (Preservation of Conversion)**.** *If* $\mathbf{e}_1 \triangleright^* \mathbf{e}_2$, *then* $\mathbf{e}_1{}^{\circ} \triangleright^* e'$ *and* $\mathbf{e}_2{}^{\circ} \equiv e'$.

Proof. By induction on the derivation of $\mathbf{e}_1 \triangleright^* \mathbf{e}_2$.[1] The proof is completely straightforward; I give the key case.

Case: Rule Red-Cong-Cont

We have that $\mathbf{e}_1\ @\ \mathbf{A}\ \lambda\, \mathbf{x} : \mathbf{B}.\, \mathbf{e}_2 \triangleright^* \mathbf{e}_1'\ @\ \mathbf{A}'\ \lambda\, \mathbf{x} : \mathbf{B}'.\, \mathbf{e}_2'$, and we must show that $let\, x = \mathbf{e}_1{}^{\circ}\ \mathbf{B}^{\circ}\ id : \mathbf{B}^{\circ}\, in\, \mathbf{e}_2{}^{\circ} \triangleright^* e'$ where $e' \equiv let\, x = \mathbf{e}_1'{}^{\circ}\ \mathbf{B}'^{\circ}\ id : \mathbf{B}'^{\circ}\, in\, \mathbf{e}_2'{}^{\circ}$.

By the induction hypothesis, we know that $\mathbf{e}_1{}^{\circ} \triangleright^* e_1 \equiv \mathbf{e}_1'{}^{\circ}$, $\mathbf{B}^{\circ} \triangleright^* B \equiv \mathbf{B}'^{\circ}$, and $\mathbf{e}_2{}^{\circ} \triangleright^* e_2 \equiv \mathbf{e}_2'{}^{\circ}$.

Therefore, by the congruence rules for extensional CoC Rule Red-Cong-Let and Rule Red-Cong-App, we know that $let\, x = \mathbf{e}_1{}^{\circ}\ \mathbf{B}^{\circ}\ id : \mathbf{B}^{\circ}\, in\, \mathbf{e}_2{}^{\circ} \triangleright^* let\, x = e_1\ B\ id : B\, in\, e_2$.

The goal follows by transitivity of $\equiv$, since

$$let\, x = e_1\ B\ id : B\, in\, e_2 \equiv let\, x = \mathbf{e}_1'{}^{\circ}\ \mathbf{B}'^{\circ}\ id : \mathbf{B}'^{\circ}\, in\, \mathbf{e}_2'{}^{\circ}$$

□

1 In the previous version of this work (Bowman et al., 2018), this proof was incorrectly stated as by induction on the length of reduction sequences.

Lemma 6.4.7 (Preservation of Equivalence). *If* $\mathbf{e}_1 \equiv \mathbf{e}_2$, *then* $\mathbf{e}_1{}^\circ \equiv \mathbf{e}_2{}^\circ$

Finally, I can show type preservation, which completes the proof of consistency. Since the translation is homomorphic on all typing rules except Rule T-Cont, there is only one interesting case in the proof of Lemma 6.4.8. We must show that Rule Un-Cont is type preserving. Note that the case for Rule Conv appeals to Lemma 6.4.7.

Lemma 6.4.8 (Type Preservation). *If* $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$ *then* $\mathbf{\Gamma}^\circ \vdash \mathbf{e}^\circ : \mathbf{A}^\circ$

Proof. By induction on the derivation $\mathbf{\Gamma} \vdash \mathbf{e} : \mathbf{A}$. There is one interesting case.

Case: Rule T-Cont

We have the following.

$$\frac{\mathbf{\Gamma} \vdash \mathbf{e}_1 : \mathbf{\Pi}\, \alpha : \star.\, (\mathbf{B} \rightarrow \alpha) \rightarrow \alpha \qquad \mathbf{\Gamma} \vdash \mathbf{A} : \star \qquad \mathbf{\Gamma}, \mathbf{x}' = \mathbf{e}_1\ \mathbf{B}\ \mathbf{id} \vdash \mathbf{e}_2 : \mathbf{A}}{\mathbf{\Gamma} \vdash \mathbf{e}_1\ @\ \mathbf{A}\ (\lambda\, \mathbf{x}' : \mathbf{B}.\, \mathbf{e}_2) : \mathbf{A}}$$

We must show $\mathbf{\Gamma}^\circ \vdash \mathit{let}\, x' = (\mathbf{e}_1{}^\circ\ \mathbf{B}^\circ\ \mathit{id})\, \mathit{in}\, \mathbf{e}_2{}^\circ : \mathbf{A}^\circ$.

By Rule Let in Extensional CoC, it suffices to show

- $\mathbf{\Gamma}^\circ \vdash (\mathbf{e}_1{}^\circ\ \mathbf{B}^\circ\ \mathit{id}) : \mathbf{B}^\circ$, which follows easily by the induction hypothesis applied to the premises of Rule T-Cont.
- $\mathbf{\Gamma}^\circ, x' = (\mathbf{e}_1{}^\circ\ \mathbf{B}^\circ\ \mathit{id}) : \mathbf{B}^\circ \vdash \mathbf{e}_2{}^\circ : \mathbf{A}^\circ$, which follows immediately by the induction hypothesis.

□

Theorem 6.4.9 (Logical Consistency of CoC^k). *There does not exist a closed term* $\mathbf{e}$ *such that* $\cdot \vdash \mathbf{e} : \bot$.

6.5 Call-By-Name CPS Translation

I now present the call-by-name CPS translation (CPS^n) of CoC^D. There are two main differences compared to the CBN CPS translation of Barthe et al. (1999): (1) I use a locally polymorphic answer type instead of the double-negation translation, which enables proving type-preservation with dependent pairs, and (2) I use a domain-full target language, which is a necessary step to get decidable type-checking for dependently typed languages.

For CPS, it is helpful to present the translation as a relation over typing derivations instead of as a function over syntax. The CPS translation requires the type of each expression in order to produce type annotation on continuations. Furthermore, the translation is not uniformly defined on expressions: terms are translated differently than types, types are translated either using the computation translation or the value

$\boxed{\Gamma \vdash \mathsf{U} \leadsto^n_U \mathbf{U}}$

$$\frac{}{\Gamma \vdash \star \leadsto^n_U \star}\ \text{CPS}^n_U\text{-Star} \qquad \frac{}{\Gamma \vdash \square \leadsto^n_U \square}\ \text{CPS}^n_U\text{-Box}$$

Figure 6.9: CPS^n of Universes

$\boxed{\Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^n_\kappa \boldsymbol{\kappa} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash \mathsf{K}^+ : \mathsf{U}^+}$

$$\frac{}{\Gamma \vdash \star : \square \leadsto^n_\kappa \star}\ \text{CPS}^n_\kappa\text{-Ax} \qquad \frac{\Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : \mathsf{K} \vdash \mathsf{K}' : \mathsf{U}' \leadsto^n_\kappa \boldsymbol{\kappa}'}{\Gamma \vdash \Pi\,\alpha : \mathsf{K}.\, \mathsf{K}' : \mathsf{U}' \leadsto^n_\kappa \boldsymbol{\Pi\,\alpha : \kappa.\, \kappa'}}\ \text{CPS}^n_\kappa\text{-PiK}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \leadsto^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{K} : \mathsf{U} \leadsto^n_\kappa \boldsymbol{\kappa}}{\Gamma \vdash \Pi\,\mathsf{x} : \mathsf{A}.\, \mathsf{K} : \mathsf{U} \leadsto^n_\kappa \boldsymbol{\Pi\,\mathbf{x} : \mathbf{A}.\, \kappa}}\ \text{CPS}^n_\kappa\text{-PiA}$$

Figure 6.10: CPS^n of Kinds

translation depending on the derivation, and kinds and universes have separate translations. The presentation as a relation on typing derivation is verbose, but makes all of these details explicit and thus easier to follow and understand why the translation is type preserving. However, as in Section 6.2, it is also useful to abbreviate these as translations over expressions for the purposes of examples and proofs. I continue to use $\mathsf{t}^\div$ for the computation translation and t^+ for the value translation. Below I give the abbreviations for all of the translation judgments. Note that anywhere I use this notation, I require the typing derivation as an implicit parameter, since formally the translation is defined over typing derivations.

$$\begin{aligned} \mathsf{A}^\div &\overset{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash \mathsf{A} : \star \leadsto^n_{A\div} \mathbf{A} \\ \mathsf{e}^\div &\overset{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash \mathsf{e} : \mathsf{A} \leadsto^n_e \mathbf{e} \end{aligned} \qquad \begin{aligned} \mathsf{U}^+ &\overset{\text{def}}{=} \mathbf{U} \text{ where } \Gamma \vdash \mathsf{U} \leadsto^n_U \mathbf{U} \\ \mathsf{K}^+ &\overset{\text{def}}{=} \boldsymbol{\kappa} \text{ where } \Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^n_\kappa \boldsymbol{\kappa} \\ \mathsf{A}^+ &\overset{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^n_A \mathbf{A} \end{aligned}$$

I start with the simple translations. The translation on universes and kinds are essentially homomorphic on the structure of the typing derivation. I define the translation for universes CPS^n_U in Figure 6.9 and kinds CPS$^n_\kappa$ in Figure 6.10, both of which I abbreviate with $^+$. There is no separate computation translation for kinds or universes, which cannot appear as computations in CoC^D.

I give the key CPS^n translations of types in Figure 6.11; the full definition is in Appendix H Figure H.5. For types, I define a value translation CPS^n_A and a computation translation CPS$^n_{A\div}$. We need separate computation and value translations for types since we internalize the concept of evaluation at the term-level, and types describe term-level computations and term-level values. Recall that this is the call-by-name translation, so function arguments, even type-level functions, are computations. Note, therefore, that

$\boxed{\Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^n_A \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash \mathsf{A}^+ : \mathsf{K}^+}$

$$\frac{}{\Gamma \vdash \alpha : \mathsf{K} \leadsto^n_A \boldsymbol{\alpha}}\ \text{CPS}^n_A\text{-Var}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K}' \leadsto^n_{A\div} \mathbf{B}}{\Gamma \vdash \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} : \mathsf{K}' \leadsto^n_A \mathbf{\Pi\,x : A.\,B}}\ \text{CPS}^n_A\text{-Pi}$$

$$\frac{\Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K}' \leadsto^n_{A\div} \mathbf{B}}{\Gamma \vdash \Pi\,\alpha : \mathsf{K}.\,\mathsf{B} : \mathsf{K}' \leadsto^n_A \mathbf{\Pi}\,\boldsymbol{\alpha} : \boldsymbol{\kappa}.\,\mathbf{B}}\ \text{CPS}^n_A\text{-PiK}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \leadsto^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \leadsto^n_A \mathbf{B}}{\Gamma \vdash \lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} : \Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{K} \leadsto^n_A \boldsymbol{\lambda}\,\mathbf{x : A.\,B}}\ \text{CPS}^n_A\text{-Constr}$$

$$\frac{\Gamma \vdash \mathsf{A} : \Pi\,\mathsf{x} : \mathsf{B}.\,\mathsf{K} \leadsto^n_A \mathbf{A} \qquad \Gamma \vdash \mathsf{e} : \mathsf{B} \leadsto^n_e \mathbf{e}}{\Gamma \vdash \mathsf{A}\ \mathsf{e} : \mathsf{K}[\mathsf{e}/\mathsf{x}] \leadsto^n_A \mathbf{A\ e}}\ \text{CPS}^n_A\text{-AppConstr}$$

$$\frac{\Gamma \vdash \mathsf{A} : \star \leadsto^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \leadsto^n_{A\div} \mathbf{B}}{\Gamma \vdash \Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B} : \star \leadsto^n_A \mathbf{\Sigma\,x : A.\,B}}\ \text{CPS}^n_A\text{-Sig}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A} \leadsto^n_e \mathbf{e} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K}' \leadsto^n_A \mathbf{B}}{\Gamma \vdash \mathsf{let}\,\mathsf{x} = \mathsf{e} : \mathsf{A}\,\mathsf{in}\,\mathsf{B} : \mathsf{K}' \leadsto^n_A \mathbf{let\,x = e : A\,in\,B}}\ \text{CPS}^n_A\text{-Let}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \qquad \Gamma \vdash \mathsf{K} \equiv \mathsf{K}' \qquad \Gamma \vdash \mathsf{A} : \mathsf{K}' \leadsto^n_A \mathbf{A}}{\Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^n_A \mathbf{A}}\ \text{CPS}^n_A\text{-Conv} \qquad \dots$$

$\boxed{\Gamma \vdash \mathsf{A} : \star \leadsto^n_{A\div} \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash \mathsf{A}^\div : \star^+}$

$$\frac{\Gamma \vdash \mathsf{A} : \star \leadsto^n_A \mathbf{A}}{\Gamma \vdash \mathsf{A} : \star \leadsto^n_{A\div} \mathbf{\Pi}\,\boldsymbol{\alpha} : \boldsymbol{\star}.\,(\mathbf{A} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}}\ \text{CPS}^n_{A\div}\text{-Comp}$$

Figure 6.11: CPS^n of Types (excerpts)

the rule Rule CPS$^n_\kappa$-PiA uses the computation translation on the domain annotation A of $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{K}$—*i.e.*, the kind describing a type-level function that abstracts over a term of type A. Most rules are straightforward. We translate type-level variables α in-place in Rule CPS^n_A-Var. Again, since this is the CBN translation, we use the computation translation on domain annotations. Rule CPS^n_A-Constr for the value translation of type-level functions that abstract over a term, $\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, translates the domain annotation A using the computation translation. The rule for the value translation of a dependent function type, Rule CPS^n_A-Pi, translates the domain annotation A using the computation translation. This means that a function is a value when it accepts a computation as an argument. Rule CPS^n_A-SIG produces the value translation of a dependent pair type by translating both components of a pair using the computation translation. This means we consider a pair a value when it contains computations as components. Note that since the translation is defined on typing derivations, we have an explicit translation of the conversion rule Rule CPS^n_A-CONV.

There is only one rule for the computation translation of a type, Rule $\text{CPS}^n_{A\div}$-COMP, which is the polymorphic answer type translation described in Section 6.2. Notice that Rule $\text{CPS}^n_{A\div}$-COMP is defined only for types of kind $\star$, since only types of kind $\star$ have inhabitants in CoC^D. For example, we cannot apply Rule $\text{CPS}^n_{A\div}$-COMP to type-level function since no term inhabits a type-level function.

The main CPS^n translation rules for terms are given in Figure 6.12; the full definition is in Appendix H Figure H.6 and Figure H.7. Intuitively, we translate each term e of type A to a term $\mathbf{e}$ of type $\mathbf{\Pi\,\alpha : \star.\,(A \to \alpha) \to \alpha}$, where $\mathbf{A}$ is the value translation of A. This type represents a computation that, when given a continuation $\mathbf{k}$ that expects a value of type $\mathbf{A}$, promises to call $\mathbf{k}$ with a value of type $\mathbf{A}$. Since we have only two value forms in the call-by-name translation, we do not explicitly define a separate value translation for terms, but inline that translation. Note that the value cases, Rule CPS^n_e-FUN and Rule CPS^n_e-PAIR, feature the same pattern—*i.e.*, produce a computation $\mathbf{\lambda\,\alpha.\,\lambda\,k.\,k\ v}$ that expects a continuation and then immediately calls that continuation on the value $\mathbf{v}$. In the case of Rule CPS^n_e-FUN, the value $\mathbf{v}$ is the function $\mathbf{\lambda\,x : A.\,e}$ produced by translating the source function $\mathsf{\lambda\,x : A.\,e}$ using the computation type translation from $\mathsf{A} \rightsquigarrow^n_{A\div} \mathbf{A}$ and the computation term translation $\mathsf{e} \rightsquigarrow^n_e \mathbf{e}$. In the case of Rule CPS^n_e-PAIR, the value we produce $\langle \mathbf{e_1}, \mathbf{e_2} \rangle$ contains computations, not values.

The rest of the term translation rules are for computations. Notice that while all terms produced by the term translation have a computation type, all continuations take a value type. Since this is a CBN translation, we consider variables as computations in Rule CPS^n_e-VAR. We translate term variables as an η-expansion of a CPS'd computation. We must η-expand the variable case to guarantee CBN evaluation order, as I discuss shortly. In Rule CPS^n_e-APP, we encode the CBN evaluation order for function application $\mathsf{e\ e'}$ in the usual way. We translate the computations $\mathsf{e} \rightsquigarrow^n_e \mathbf{e}$ and $\mathsf{e'} \rightsquigarrow^n_e \mathbf{e'}$. First, we evaluate $\mathbf{e}$ to a value $\mathbf{f}$, then apply $\mathbf{f}$ to the *computation* $\mathbf{e'}$. The application $\mathbf{f\ e'}$ is itself a computation, which we call with the continuation $\mathbf{k}$.

Notice that only the translation rules Rule CPS^n_e-FST and Rule CPS^n_e-SND use the new $\mathbf{@}$ form. As discussed in, Section 6.2, to type check the translation of $\mathsf{snd\,e}$ produced by Rule CPS^n_e-SND, we require the rule Rule T-CONT when type checking the continuation that performs the second projection. While type checking the continuation, we know that the value $\mathbf{y}$ that the continuation receives is equivalent to $\mathsf{e}^{\div}\ \mathbf{\alpha\ id}$. The reason we must use the $\mathbf{@}$ syntax in the in the translation Rule CPS^n_e-FST, whose type is not dependent, is so that we can apply the Rule $\equiv$-CONT rule to resolve the equivalence of the two *first projections* in the type of the second projection. That is, as we saw in Section 6.2, type preservation fails because we must show equivalence between $(\mathsf{fst\,e})^{\div}$ and $\mathbf{fst\,y}$. Since these are the only two cases that require our new

$\boxed{\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash e^\div : A^\div}$

$$\dfrac{\Gamma \vdash A : K \rightsquigarrow^n_A \mathbf{A}}{\Gamma \vdash x : A \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \to \alpha.\, x\ \alpha\ k}}\ \text{CPS}^n_e\text{-Var}$$

$$\dfrac{\Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : K' \rightsquigarrow^n_{A\div} \mathbf{B} \qquad \Gamma, x : A \vdash e : B \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash \lambda x : A.\, e : \Pi x : A.\, B \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : (\Pi\,x : A.\,B) \to \alpha.\, k\ (\lambda\,x : A.\, e)}}\ \text{CPS}^n_e\text{-Fun}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash e : \Pi x : A.\, B \rightsquigarrow^n_e \mathbf{e} \qquad \Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A^\div} \\ \Gamma, x : A \vdash B : K' \rightsquigarrow^n_{A\div} \mathbf{B^\div} \qquad \Gamma, x : A \vdash B : K' \rightsquigarrow^n_A \mathbf{B^+} \qquad \Gamma \vdash e' : A \rightsquigarrow^n_e \mathbf{e'}\end{array}}{\begin{array}{r}\Gamma \vdash e\ e' : B[e'/x] \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : (B^+[e'/x]) \to \alpha.} \\ \boldsymbol{e\ \alpha\ (\lambda\,f : \Pi\,x : A^\div.\,B^\div.\,(f\ e')\ \alpha\ k)}\end{array}}\ \text{CPS}^n_e\text{-App}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash e_1 : A \rightsquigarrow^n_e \mathbf{e_1} \\ \Gamma \vdash e_2 : B[e_1/x] \rightsquigarrow^n_e \mathbf{e_2} \qquad \Gamma \vdash A : \star \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow^n_{A\div} \mathbf{B}\end{array}}{\begin{array}{r}\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma x : A.\, B \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : \Sigma\,x : A.\,B \to \alpha.} \\ \boldsymbol{k\ \langle e_1, e_2 \rangle\ \text{as}\ \Sigma\,x : A.\,B}\end{array}}\ \text{CPS}^n_e\text{-Pair}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash A : \star \rightsquigarrow^n_{A\div} \mathbf{A^\div} \\ \Gamma, x : A \vdash B : \star \rightsquigarrow^n_{A\div} \mathbf{B^\div} \qquad \Gamma \vdash A : \star \rightsquigarrow^n_A \mathbf{A^+} \qquad \Gamma \vdash e : \Sigma x : A.\, B \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{r}\Gamma \vdash \mathsf{fst}\, e : A \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A^+ \to \alpha.} \\ \boldsymbol{e\ @\ \alpha\ (\lambda\,y : \Sigma\,x : A^\div.\,B^\div.\ \text{let}\ z = \text{fst}\ y\ \text{in}\ z\ \alpha\ k)}\end{array}}\ \text{CPS}^n_e\text{-Fst}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash A : \star \rightsquigarrow^n_{A\div} \mathbf{A^\div} \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow^n_{A\div} \mathbf{B^\div} \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow^n_A \mathbf{B^+} \\ \Gamma \vdash (\mathsf{fst}\, e) : A \rightsquigarrow^n_e (\mathsf{fst}\, e)^\div \qquad \Gamma \vdash e : \Sigma x : A.\, B \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{r}\Gamma \vdash \mathsf{snd}\, e : B[(\mathsf{fst}\, e)/x] \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : B^+}[(\mathsf{fst}\, e)^\div/\mathbf{x}] \boldsymbol{\to \alpha.} \\ \boldsymbol{e\ @\ \alpha\ (\lambda\,y : \Sigma\,x : A^\div.\,B^\div.} \\ \boldsymbol{\text{let}\ z = \text{snd}\ y\ \text{in}\ z\ \alpha\ k)}\end{array}}\ \text{CPS}^n_e\text{-Snd}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e} \qquad \Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A} \\ \Gamma, x = e : A \vdash B : K' \rightsquigarrow^n_A \mathbf{B} \qquad \Gamma, x = e : A \vdash e' : B \rightsquigarrow^n_e \mathbf{e'}\end{array}}{\begin{array}{r}\Gamma \vdash \mathsf{let}\, x = e : A\ \mathsf{in}\ e' : B[e/x] \rightsquigarrow^n_e \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : B[e/x] \to \alpha.} \\ \boldsymbol{\text{let}\ x = e : A\ \text{in}\ e'\ \alpha\ k}\end{array}}\ \text{CPS}^n_e\text{-Let}$$

$$\dfrac{\Gamma \vdash e : B \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e}}\ \text{CPS}^n_e\text{-Conv} \qquad\qquad \ldots$$

Figure 6.12: CPS^n of Terms (excerpts)

rules, these are the only cases where we need to use the @ form in the translation; all other translation rules use standard application. In Section 6.6, I show that that the CBV translation must use the @ form much more frequently since, intuitively, the new

$\boxed{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \text{ Lemma 6.5.5 will show } \vdash \mathbf{\Gamma}^+}$

$$\frac{}{\vdash \cdot \rightsquigarrow^n \cdot}\ \text{CPS}^n_\Gamma\text{-Empty} \qquad \frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A}}{\vdash \Gamma, x : A \rightsquigarrow^n \mathbf{\Gamma}, \mathbf{x} : \mathbf{A}}\ \text{CPS}^n_\Gamma\text{-AssumT}$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash K : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha : K \rightsquigarrow^n \mathbf{\Gamma}, \boldsymbol{\alpha} : \boldsymbol{\kappa}}\ \text{CPS}^n_\Gamma\text{-AssumK}$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e}}{\vdash \Gamma, x = e : A \rightsquigarrow^n \mathbf{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}}\ \text{CPS}^n_\Gamma\text{-Def}$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \mathbf{\Gamma} \qquad \Gamma \vdash A : K \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash K : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha = A : K \rightsquigarrow^n \mathbf{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}}\ \text{CPS}^n_\Gamma\text{-DefT}$$

Figure 6.13: CPS^n of Environments

equivalence rule recovers a notion of "value" in the CPS'd language, and in call-by-*value* types can only depend on values.

The CPS translation encodes the CBN evaluation order explicitly so that the evaluation order of compiled terms is independent of the target language's evaluation order. This property is not immediately obvious since Rule CPS^n_e-Let binds a variable $\mathbf{x}$ to an expression $\mathbf{e}$, making it seem like there are two possible evaluation orders: either evaluate $\mathbf{e}$ first, or substitute $\mathbf{e}$ for $\mathbf{x}$ first. Note, however, that the CBN translation always produces a $\boldsymbol{\lambda}$ term—even in the variable case since Rule CPS^n_e-Var employs η-expansion as noted above. Therefore, in Rule CPS^n_e-Let, $\mathbf{e}$ will always be a value, which means it doesn't evaluate in either CBN or CBV. Therefore, there is no ambiguity in how to evaluate the translation of let.

The translation rule Rule CPS^n_e-Conv is deceptively simple. We could equivalently write this translation as follows, which makes its subtlety apparent.

$$\frac{\begin{array}{c}\Gamma \vdash e : B \\ \Gamma \vdash A \equiv B \qquad \Gamma \vdash e : B \rightsquigarrow^n_e \mathbf{e} \qquad \Gamma \vdash A : K \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash B : K \rightsquigarrow^n_A \mathbf{B}\end{array}}{\Gamma \vdash e : A \rightsquigarrow^n_e \boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\,\boldsymbol{\lambda}\,\mathbf{k} : \mathbf{A} \to \boldsymbol{\alpha}.\,\mathbf{e}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{k}\ \mathbf{x})}\ \text{CPS}^n_e\text{-Conv}$$

Notice now that while the continuation $\mathbf{k}$ expects a term of type $\mathbf{A}$, we call $\mathbf{k}$ with a term of type $\mathbf{B}$. Intuitively, $\mathbf{k}\ \mathbf{x}$ should be well-typed since $\mathbf{A}$ and $\mathbf{B}$ should be equivalent. However, recall from Chapter 3 that we cannot prove that $\mathbf{A}$ and $\mathbf{B}$ are equivalence until we prove compositionality, which requires that we appeal to this translation rule. As a result, we require equivalence to be non-type-directed.

I lift the translations to typing environments in the usual way in Figure 6.13. Since this is the CBN translation, we recur over the environment applying the computation translation.

6.5.1 Type Preservation

The proof of type preservation follows the standard architecture from Chapter 3.

First I show Lemma 6.5.1 (CPS^n Compositionality), which states that the CPS^n translation commutes with substitution. The formal statement of the lemma is somewhat complicated since we have the cross product of four syntactic categories and two translations. However, the intuition is simple: first substituting and then translating is equivalent to translating and then substituting.

Lemma 6.5.1 (CPS^n Compositionality)**.**

1. $(\mathsf{K}[\mathsf{A}/\alpha])^+ \equiv \mathsf{K}^+[\mathsf{A}^+/\boldsymbol{\alpha}]$
2. $(\mathsf{K}[\mathsf{e}/\mathsf{x}])^+ \equiv \mathsf{K}^+[\mathsf{e}^\div/\mathbf{x}]$
3. $(\mathsf{A}[\mathsf{B}/\alpha])^+ \equiv \mathsf{A}^+[\mathsf{B}^+/\boldsymbol{\alpha}]$
4. $(\mathsf{A}[\mathsf{e}/\mathsf{x}])^+ \equiv \mathsf{A}^+[\mathsf{e}^\div/\mathbf{x}]$
5. $(\mathsf{A}[\mathsf{B}/\alpha])^\div \equiv \mathsf{A}^\div[\mathsf{B}^+/\boldsymbol{\alpha}]$
6. $(\mathsf{A}[\mathsf{e}/\mathsf{x}])^\div \equiv \mathsf{A}^\div[\mathsf{e}^\div/\mathbf{x}]$
7. $(\mathsf{e}[\mathsf{A}/\alpha])^\div \equiv \mathsf{e}^\div[\mathsf{A}^+/\boldsymbol{\alpha}]$
8. $(\mathsf{e}[\mathsf{e}'/\mathsf{x}])^\div \equiv \mathsf{e}^\div[\mathsf{e}'^\div/\mathbf{x}]$

Proof. In the PTS syntax, we represent source expressions as $\mathsf{t}[\mathsf{t}'/\mathsf{x}]$. The proof is by induction on the typing derivations for t. Note that our $^\div$ and $^+$ notation implicitly require the typing derivations as premises. The proof is completely straightforward. Part 6 follows immediately by part 3. Part 7 follows immediately by part 4. I give representative cases for the other parts.

Case: Rule *, parts 1 and 2. Trivial, since no free variables appear in $\star$.

Case: Rule Pi-* and Rule Pi-□: $\mathsf{t} = \Pi\,\mathsf{x} : \mathsf{B}.\,\mathsf{K}'$

Sub-case: Part 1. We must show that $((\Pi\,\mathsf{x} : \mathsf{B}.\,\mathsf{K}')[\mathsf{A}/\alpha])^+ = (\Pi\,\mathsf{x} : \mathsf{B}.\,\mathsf{K}')^+[\mathsf{A}^+/\boldsymbol{\alpha}]$.

$$
\begin{aligned}
& ((\Pi\,\mathsf{x} : \mathsf{B}.\,\mathsf{K}')[\mathsf{A}/\alpha])^+ \\
= {} & (\Pi\,\mathsf{x} : \mathsf{B}[\mathsf{A}/\alpha].\,\mathsf{K}'[\mathsf{A}/\alpha])^+ && (90) \\
& \text{by definition of substitution} \\
= {} & \boldsymbol{\Pi}\,\mathbf{x}' : (\mathsf{B}[\mathsf{A}/\alpha])^\div.\,(\mathsf{K}'[\mathsf{A}/\alpha])^+ && (91) \\
& \text{by definition of the translation} \\
= {} & \boldsymbol{\Pi}\,\mathbf{x}' : \mathsf{B}^\div[\mathsf{A}^+/\alpha].\,\mathsf{K}'^+[\mathsf{A}^+/\alpha] && (92) \\
& \text{by parts 1 and 5 of the induction hypothesis} \\
= {} & (\boldsymbol{\Pi}\,\mathbf{x}' : \mathsf{B}^\div.\,\mathsf{K}'^+)[\mathsf{A}^+/\boldsymbol{\alpha}] && (93) \\
& \text{by definition of substitution} \\
= {} & (\Pi\,\mathsf{x}' : \mathsf{B}.\,\mathsf{K}')^+[\mathsf{A}^+/\boldsymbol{\alpha}] && (94) \\
& \text{by definition of the translation}
\end{aligned}
$$

Sub-case: Part 2. Similar to the previous sub-case.

Case: Rule Var

Sub-case: Part 3 $t = \alpha'$. Part 4 is trivial since x is not free in α.

We must show that $(\alpha'[A/\alpha])^+ = \boldsymbol{\alpha}'[A^+/\boldsymbol{\alpha}]$.

Sub-sub-case: $\alpha = \alpha'$. It suffices to show that $A^+ = A^+$, which is trivial.

Sub-sub-case: $\alpha \neq \alpha'$. $\alpha^+ = \boldsymbol{\alpha}$ by definition.

Sub-case: x, parts 7 and 8. Similar to previous case.

Case: Rule App

Sub-case: $t = e_1\ e_2$, Part 7

We must show $((e_1\ e_2)[A'/\alpha'])^{\div} = (e_1\ e_2)^{\div}[A'^+/\boldsymbol{\alpha}']$.

$$\begin{aligned}
& ((e_1\ e_2)[A'/\alpha'])^{\div} \\
= \; & (e_1[A/\alpha']\ e_2[A'/\alpha'])^{\div} && (95) \\
& \text{by definition of substitution} \\
= \; & \boldsymbol{\lambda\,\alpha} : \star.\,\boldsymbol{\lambda\,k} : ((B[A'/\alpha'])^+[(e_2[A'/\alpha'])^{\div}/\mathbf{x}]) \rightarrow \boldsymbol{\alpha}. && (96) \\
& \quad (e_1[A'/\alpha'])^{\div}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f} : \boldsymbol{\Pi\,x} : (A[A'/\alpha'])^{\div}.\,(B[A'/\alpha'])^{\div}. \\
& \qquad\qquad (\mathbf{f}\ (e_2[A'/\alpha'])^{\div})\ \boldsymbol{\alpha}\ \mathbf{k}) \\
& \text{by def. of translation} \\
= \; & \boldsymbol{\lambda\,\alpha} : \star.\,\boldsymbol{\lambda\,k} : (B^+[A'^+/\boldsymbol{\alpha}'][e_2^{\div}[A'^+/\boldsymbol{\alpha}']/\mathbf{x}]) \rightarrow \boldsymbol{\alpha}. && (97) \\
& \quad e_1^{\div}[A'^+/\boldsymbol{\alpha}']\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f} : \boldsymbol{\Pi\,x} : A^{\div}[A'^+/\boldsymbol{\alpha}'].\,B^{\div}[A'^+/\boldsymbol{\alpha}']. \\
& \qquad\qquad (\mathbf{f}\ e_2^{\div}[A'^+/\boldsymbol{\alpha}'])\ \boldsymbol{\alpha}\ \mathbf{k}) \\
& \text{by part 3, 5, and 7 of IH} \\
= \; & (\boldsymbol{\lambda\,\alpha} : \star.\,\boldsymbol{\lambda\,k} : (B^+[e_2^{\div}/\mathbf{x}]) \rightarrow \boldsymbol{\alpha}. && (98) \\
& \quad e_1^{\div}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f} : \boldsymbol{\Pi\,x} : A^{\div}.\,B^{\div}. \\
& \qquad\qquad (\mathbf{f}\ e_2^{\div})\ \boldsymbol{\alpha}\ \mathbf{k}))[A'^+/\boldsymbol{\alpha}'] \\
& \text{by definition of substitution} \\
= \; & (e_1\ e_2)^{\div}[A'^+/\boldsymbol{\alpha}'] && (99) \\
& \text{by definition of translation}
\end{aligned}$$

Sub-case: Part 8

We must show $((e_1\ e_2)[e/x])^{\div} = (e_1\ e_2)^{\div}[e^{\div}/\mathbf{x}]$.

Similar to previous case.

Case: Rule Conv. The proof is trivial, now that we have staged the proof appropriately. I give part 8 as an example.

$$\frac{\Gamma \vdash e : B \qquad \Gamma \vdash A : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : A}$$

We must show that $(e[e'/x])^{\div} \equiv e^{\div}[e'^{\div}/\mathbf{x}]$ (at type A). Note that by part 8 of the induction hypothesis, we know that $(e[e'/x])^{\div} \equiv e^{\div}[e'^{\div}/\mathbf{x}]$ (at the smaller derivation for type B). But recall that equivalence is not type directed, so the proof is complete. □

I next prove that the translation preserves reduction and conversion, Lemma 6.5.2 and Lemma 6.5.3. Note that kinds cannot take steps in the reduction relation, but can in the conversion relation since it reduces under all contexts. Note that we can only preserve reduction and conversion up to equivalence, in particular η-equivalence. The intuition for this is simple. The computation translation of a term $e'^{\div}$ always produce a λ-expression $\boldsymbol{\lambda\,\alpha.\,\lambda\,k.\,e''}$. However, when $e^{\div} \triangleright^{*} \mathbf{e'}$, we do not know that the term $\mathbf{e'}$ is equal to a λ-expression, although it is η-equivalent to one.

Lemma 6.5.2 (CPS^n Preservation of Reduction)**.**

- *If* $\Gamma \vdash e : A$ *and* $e \triangleright e'$ *then* $e^{\div} \triangleright^{*} \mathbf{e'}$ *and* $\mathbf{e'} \equiv e'^{\div}$
- *If* $\Gamma \vdash A : K$ *and* $A \triangleright A'$ *then* $A^{+} \triangleright^{*} \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^{+}$
- *If* $\Gamma \vdash A : \star$ *and* $A \triangleright A'$ *then* $A^{\div} \triangleright^{*} \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^{\div}$

Proof. The proof is straightforward by cases on the reduction relation. I give some representative cases.

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

It suffices to show that $x^{\div} \triangleright_{\delta} e'^{\div}$ where $x^{\div} = e'^{\div} : A'^{\div} \in \Gamma^{+}$, which follows by Rule CPS$^{n}_{\Gamma}$-Def.

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

We must show that $((\lambda\, x : \_.\, e_1)\ e_2)^{\div} \triangleright^{*} \mathbf{e'}$ and $\mathbf{e'} \equiv (e_1[e_2/x])^{\div}$.

$$\begin{aligned}
& ((\lambda\, x : \_.\, e_1)\ e_2) \\
= \;& \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : \_.} \qquad (100)\\
& \quad (\boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : \_.\,k}\ (\boldsymbol{\lambda\,x : \_.}\, e_1^{\div}))\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f : \_.}\,(\mathbf{f}\ e_2^{\div})\ \boldsymbol{\alpha}\ \mathbf{k}) \\
& \text{by definition of translation} \\
\triangleright^{*}\; & \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : \_.}\,(((\boldsymbol{\lambda\,x : \_.}\, e_1^{\div})\ e_2^{\div})\ \boldsymbol{\alpha}\ \mathbf{k}) \qquad (101)\\
& \text{by } \triangleright_{\beta} \\
\triangleright^{*}\; & \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : \_.}\,(e_1^{\div}[e_2^{\div}/\mathbf{x}])\ \boldsymbol{\alpha}\ \mathbf{k} \qquad (102)\\
\equiv\; & e_1^{\div}[e_2^{\div}/\mathbf{x}] \qquad (103)\\
& \text{by Rule } \equiv\text{-}\eta \\
= \;& (e_1[e_2/x])^{\div} \qquad (104)\\
& \text{by Lemma 6.5.1}
\end{aligned}$$

Case: $\mathsf{snd}\ \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \rhd_{\pi_2} \mathsf{e}_2$

We must show that $(\mathsf{snd}\ \langle \mathsf{e}_1, \mathsf{e}_2 \rangle)^{\div} \rhd^* \mathbf{e'}$ and $\mathbf{e'} \equiv \mathsf{e}_2^{\div}$.

$$
\begin{aligned}
& (\mathsf{snd}\ \langle \mathsf{e}_1, \mathsf{e}_2 \rangle)^{\div} \\
= \;& \boldsymbol{\lambda\, \alpha : \star. \lambda\, k : \_.} && (105)\\
& \quad (\boldsymbol{\lambda\, \alpha : \star. \lambda\, k : \_.\, k}\ \langle \mathsf{e}_1^{\div}, \mathsf{e}_2^{\div} \rangle)\ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, y : \_.\, \mathbf{let}\, z = \mathbf{snd}\, y\, \mathbf{in}\, z\ \alpha\ k}) \\
\rhd^* \;& \boldsymbol{\lambda\, \alpha : \star. \lambda\, k : \_.\, \mathbf{let}\, z = \mathbf{snd}}\ \langle \mathsf{e}_1^{\div}, \mathsf{e}_2^{\div} \rangle\ \boldsymbol{\mathbf{in}\, z\ \alpha\ k} && (106)\\
\rhd^* \;& \boldsymbol{\lambda\, \alpha : \star. \lambda\, k : \_.}\, \mathsf{e}_2^{\div}\ \boldsymbol{\alpha\ k} && (107)\\
\equiv \;& \mathsf{e}_2^{\div} \quad \text{by Rule } \equiv\text{-}\eta && (108)
\end{aligned}
$$

□

Lemma 6.5.3 (CPS^n Preservation of Conversion)**.**

- *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $\mathsf{e} \rhd^* \mathsf{e}'$ *then* $\mathsf{e}^{\div} \rhd^* \mathbf{e'}$ *and* $\mathbf{e'} \equiv \mathsf{e}'^{\div}$
- *If* $\Gamma \vdash \mathsf{A} : \mathsf{K}$ *and* $\mathsf{A} \rhd^* \mathsf{A}'$ *then* $\mathsf{A}^{+} \rhd^* \mathbf{A'}$ *and* $\mathbf{A'} \equiv \mathsf{A}'^{+}$
- *If* $\Gamma \vdash \mathsf{A} : \star$ *and* $\mathsf{A} \rhd^* \mathsf{A}'$ *then* $\mathsf{A}^{\div} \rhd^* \mathbf{A'}$ *and* $\mathbf{A'} \equiv \mathsf{A}'^{\div}$
- *If* $\Gamma \vdash \mathsf{K} : \mathsf{U}$ *and* $\mathsf{K} \rhd^* \mathsf{K}'$ *then* $\mathsf{K}^{+} \rhd^* \boldsymbol{\kappa'}$ *and* $\boldsymbol{\kappa'} \equiv \mathsf{K}'^{+}$

Proof. The proof is straightforward by induction on the conversion derivation, *i.e.*, on $\Gamma \vdash \mathsf{t} \rhd^* \mathsf{t}'$.[2] □

Lemma 6.5.4 (CPS^n Preservation of Equivalence)**.**

- *If* $\mathsf{e} \equiv \mathsf{e}'$ *then* $\mathsf{e}^{\div} \equiv \mathsf{e}'^{\div}$
- *If* $\mathsf{A} \equiv \mathsf{A}'$ *then* $\mathsf{A}^{\div} \equiv \mathsf{A}'^{\div}$
- *If* $\mathsf{A} \equiv \mathsf{A}'$ *then* $\mathsf{A}^{+} \equiv \mathsf{A}'^{+}$
- *If* $\mathsf{K} \equiv \mathsf{K}'$ *then* $\mathsf{K}^{+} \equiv \mathsf{K}'^{+}$

Proof. The proof is by induction on the derivation of $\mathsf{e} \equiv \mathsf{e}'$. The base case follows by Lemma 6.5.3, and the cases of η-equivalence follow from Lemma 6.5.3, the induction hypothesis, and the fact that we have the same η-equivalence rules in the CoC^k. □

I first prove type and well-formedness preservation, Lemma 6.5.5, using the explicit syntax on which I defined CPS^n. In this lemma, proving that the translation of $\mathsf{snd}\,\mathsf{e}$ preserves typing requires both the new typing rule Rule T-Cont and the equivalence rule Rule $\equiv$-Cont. The rest of the proof is straightforward.

Lemma 6.5.5 (CPS^n Type and Well-formedness Preservation)**.**

2 In the previous version of this work (Bowman et al., 2018), this proof was incorrectly stated as by induction on the length of reduction sequences.

1. *If* $\vdash \Gamma$ *then* $\vdash \Gamma^+$
2. *If* $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash e^\div : A^\div$
3. *If* $\Gamma \vdash A : K$ *then* $\Gamma^+ \vdash A^+ : K^+$
4. *If* $\Gamma \vdash A : \star$ *then* $\Gamma^+ \vdash A^\div : \star^+$
5. *If* $\Gamma \vdash K : U$ *then* $\Gamma^+ \vdash K^+ : U^+$

Proof. All cases are proven simultaneously by simultaneous induction on the type derivation and well-formedness derivation. Part 4 follows easily by part 3 in every case, so I elide its proof. Most cases follow easily from the induction hypotheses.

Case: Part 5, Rule *: $\Gamma \vdash \star : \Box$

We must show that $\Gamma^+ \vdash \star^+ : \Box^+$, which follows by part 1 of the induction hypothesis and by definition of the translation, since $\star^+ = \star$ and $\Box^+ = \Box$.

Case: Rule Pi-*: $\Gamma \vdash \Pi\, x : e_1 .\, e_2 : K$

There are two sub-cases: either e_2 is a type, or a kind.

Sub-case: Part 3, $e_2 = B$, *i.e.*, is a type.

There are two sub-cases: either e_1 is a type or a kind.

Sub-sub-case: $e_1 = A$, *i.e.*, is a type.

We have $\Gamma \vdash \Pi\, x : A.\, B : \star$.

We must show that $\Gamma^+ \vdash (\Pi\, x : A.\, B)^+ : \star^+$

By definition, must show $\Gamma^+ \vdash \boldsymbol{\Pi\, x} : A^\div .\, B^\div : \star$, which follows by the part 4 of the induction hypothesis applied to A and B.

Sub-sub-case: $e_1 = K$, *i.e.*, is a kind.

We have $\Gamma \vdash \Pi\, \alpha : K.\, B : \star$.

We must show that $\Gamma^+ \vdash (\Pi\, \alpha : K.\, B)^+ : \star^+$

By definition, must show $\Gamma^+ \vdash \boldsymbol{\Pi\, \alpha} : K^+ .\, B^\div : \star$, which follows by the part 4 of the induction hypothesis applied to B, and part 5 of the induction hypothesis applied to K.

Sub-case: Part 5, $e_2 = K'$, *i.e.*, is a kind.

There are two sub-cases: either e_1 is a type or a kind.

Sub-sub-case: $e_1 = A$, *i.e.*, is a type.

We have $\Gamma \vdash \Pi\, x : A.\, K' : U$.

We must show that $\Gamma^+ \vdash (\Pi\, x : A.\, K')^+ : \star^+$

By definition, must show $\Gamma^+ \vdash \boldsymbol{\Pi\, x} : A^\div .\, K'^+ : \star$, which follows by the part 4 of the induction hypothesis applied to A and part 5 of the induction hypothesis applied to K.

Sub-sub-case: $e_1 = K$, *i.e.*, is a kind.

We have $\Gamma \vdash \Pi\, \alpha : K.\, K' : \star$.

We must show that $\Gamma^+ \vdash (\Pi\, \alpha : K.\, K')^+ : \star^+$

By definition, must show $\Gamma^+ \vdash \boldsymbol{\Pi}\, \boldsymbol{\alpha} : K^+.\, K'^+ : \boldsymbol{\star}$, which follows by the part 5 of the induction hypothesis applied to K and K'^+.

Case: Rule PI-□ Similar to the previous case, except with $\star$ replaced by $\square$; there are two fewer cases since this must be a kind.

Case: Rule SIG $\Gamma \vdash \Sigma\, x : A.\, B : \star$

We must show that $\Gamma^+ \vdash \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div} : \boldsymbol{\star}$, which follows easily by the part 4 of the induction hypothesis applied to A and B.

Case: Rule PAIR $\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma\, x : A.\, B$

By definition of the translation, we must show that

$$\Gamma^+ \vdash \boldsymbol{\lambda}\, \boldsymbol{\alpha} : \boldsymbol{\star}.\, \boldsymbol{\lambda}\, \mathbf{k} : (\boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div} \rightarrow \boldsymbol{\alpha}).$$
$$\mathbf{k}\ \langle e_1^{\div}, e_2^{\div} \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div} : \boldsymbol{\Pi}\, \boldsymbol{\alpha} : \boldsymbol{\star}.\, (\boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$$

It suffices to show that $\Gamma^+ \vdash \langle e_1^{\div}, e_2^{\div} \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div} : \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div}$, which follows easily by part 2 of the induction hypothesis applied to $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : B[e_1/x]$.

Case: Rule SND $\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x]$

We must show that

$$\boldsymbol{\lambda}\, \boldsymbol{\alpha} : \boldsymbol{\star}.\, \boldsymbol{\lambda}\, \mathbf{k} : B^+[(\mathsf{fst}\, e)^{\div}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}.$$
$$e^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div}.\ \mathbf{let}\ \mathbf{z} = \mathbf{snd}\ \mathbf{y}\ \mathbf{in}\ \mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k})$$

has type $(B[\mathsf{fst}\, e/x])^{\div}$.

By part 6 of Lemma 6.5.1, and definition of the translation, this type is equivalent to

$$\boldsymbol{\Pi}\, \boldsymbol{\alpha} : \boldsymbol{\star}.\, (B^+[(\mathsf{fst}\, e)^{\div}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$$

By Rule LAM, it suffices to show that

$$\Gamma^+, \boldsymbol{\alpha} : \boldsymbol{\star}, \mathbf{k} : B^+[(\mathsf{fst}\, e)^{\div}/x] \rightarrow \boldsymbol{\alpha} \vdash e^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div}.$$
$$\mathbf{let}\ \mathbf{z} = \mathbf{snd}\ \mathbf{y}\ \mathbf{in}\ \mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k}) : \boldsymbol{\alpha}$$

This is the key difficulty in the proof. Recall from Section 6.2 that the term $\mathbf{z}\ \boldsymbol{\alpha}$ has type $(B^+[\mathbf{fst}\ \mathbf{y}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$ while the term $\mathbf{k}$ has type $B^+[(\mathsf{fst}\, e)^{\div}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}$. To show that $\mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k}$ is well-typed, we must show that $(\mathsf{fst}\, e)^{\div} \equiv \mathbf{fst}\ \mathbf{y}$. I proceed by the new typing rule Rule T-CONT, which will help us prove this.

First, note that $e^{\div}\ (\boldsymbol{\Sigma}\, \mathbf{x} : A^{\div}.\, B^{\div})\ \mathbf{id}$ is well-typed. By part 4 of the induction hypothesis we know that $\Gamma^+ \vdash A^{\div} : \boldsymbol{\star}$ and $\Gamma^+, \mathbf{x} : A^{\div} \vdash B^{\div} : \boldsymbol{\star}$. By part 2 of the induction hypothesis applied to $\Gamma \vdash e : \Sigma x : A. B$, we know $\Gamma^+ \vdash e^\div : \Pi \alpha : \star. (\Sigma x : A^\div. B^\div \to \alpha) \to \alpha$.

Now, by Rule T-Cont, it suffices to show that

$$\Gamma^+, \alpha : \star, k : B^+[(\mathsf{fst}\, e)^\div / x] \to \alpha, y = e^\div\ \Sigma x{:}A^\div. B^\div\, \mathbf{id} \vdash \mathbf{let}\, z{=}\mathbf{snd}\, y\, \mathbf{in}\, z\ \alpha\ k : \alpha$$

Note that we now have the definitional equivalence $y = e^\div\ (\Sigma x : A^\div. B^\div)\ \mathbf{id}$. By Rule Let it suffices to show

$$\begin{aligned}\Gamma^+, \alpha : \star, k : B^+[(\mathsf{fst}\, e)^\div / x] \to \alpha, &\vdash z\ \alpha\ k : \alpha\\ y = e^\div\ \Sigma x : A^\div. B^\div\ \mathbf{id},&\\ z = \mathbf{snd}\, y : B^\div[\mathbf{fst}\, y / x]&\end{aligned}$$

Note that

$$\begin{aligned} z :\ & B^\div[\mathbf{fst}\, y / x] && && (109)\\ =\ & \Pi \alpha : \star. (B^+[\mathbf{fst}\, y / x] \to \alpha) \to \alpha && \text{by definition of } B^\div && (110)\\ \equiv\ & \Pi \alpha : \star. (B^+[\mathbf{fst}\, (e^\div\ \_\ \mathbf{id}) / x] \to \alpha) \to \alpha && \text{by } \delta \text{ reduction on } y && (111)\end{aligned}$$

Equation (111) above, in which we δ-reduce y, is impossible without Rule T-Cont.

By Rule Conv, and since $k : B^+[(\mathsf{fst}\, e)^\div / x] \to \alpha$, to show $z\ \alpha\ k : \alpha$ it suffices to show that $(\mathsf{fst}\, e)^\div \equiv \mathbf{fst}\, (e^\div\ \_\ \mathbf{id})$.

Note that $(\mathsf{fst}\, e)^\div\ = \lambda \alpha : \star. \lambda k' : (A^+ \to \alpha).$
$e^\div\ @\ \alpha\ (\lambda y : \Sigma x : A^\div. B^\div. \mathbf{let}\, z' = \mathbf{fst}\, y : A^\div\ \mathbf{in}\, z'\ \alpha\ k')$
by definition of the translation.

By Rule $\equiv$-η, it suffices to show that

$$\begin{aligned} & e^\div\ @\ \alpha\ (\lambda y : \Sigma x : A^\div. B^\div. \mathbf{let}\, z' = \mathbf{fst}\, y : A^\div\ \mathbf{in}\, z'\ \alpha\ k') && && (112)\\ \equiv\ & (\lambda y : \Sigma x : A^\div. B^\div. \mathbf{let}\, z' = \mathbf{fst}\, y\, \mathbf{in}\, z'\ \alpha\ k')\ (e^\div\ \_\ \mathbf{id}) && \text{Rule } \equiv\text{-Cont} && (113)\\ \equiv\ & (\mathbf{fst}\, (e^\div\ \_\ \mathbf{id}))\ \alpha\ k' && \text{by reduction} && (114)\end{aligned}$$

Notice that Equation (113) requires Rule $\equiv$-Cont applied to the translation of the fst.

Case: Rule Lam

I give proofs for only the term-level functions; the type-level functions follow exactly the same structure as type-level function types. There are two subcases.

Sub-case: The function abstracts over a term, $\Gamma \vdash \lambda x : A. e : \Pi x : A. B$

We must show

$\Gamma^+ \vdash (\lambda x : A. e)^\div : (\Pi x : A. B)^\div$.

By definition of the translation, we must show

$\Gamma^+ \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\, \boldsymbol{\lambda}\,\mathbf{k} : (\boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div) \rightarrow \boldsymbol{\alpha}.$
$\quad (\mathbf{k}\ (\boldsymbol{\lambda}\,\mathbf{x} : A^\div.\, e^\div)) : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\, (\boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$

It suffices to show that

$\Gamma^+, \boldsymbol{\alpha} : \star, \mathbf{k} : \boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div \rightarrow \boldsymbol{\alpha} \vdash \mathbf{k}\ (\boldsymbol{\lambda}\,\mathbf{x} : A^\div.\, e^\div) : \boldsymbol{\alpha}.$

By Rule App, it suffices to show that

$\Gamma^+, \boldsymbol{\alpha} : \star, \mathbf{k} : \boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div \rightarrow \boldsymbol{\alpha} \vdash (\boldsymbol{\lambda}\,\mathbf{x} : A^\div.\, e^\div) : \boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div$

By part 2 of the induction hypothesis applied to $\Gamma, x : A \vdash e : B$, we know that

$\Gamma^+, \mathbf{x} : A^\div \vdash e^\div : B^\div$

It suffices to show that

$\vdash \Gamma^+, \boldsymbol{\alpha} : \star, \mathbf{k} : \boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div \rightarrow \boldsymbol{\alpha}$ which follows easily by part 4 of the induction hypothesis applied to the typing derivations for A and B.

Sub-case: The function abstracts over a type, $\Gamma \vdash \lambda\,\alpha : K.\, e : \Pi\,\alpha : K.\, B$

We must show $\Gamma^+ \vdash (\lambda\,\alpha : K.\, e)^\div : (\Pi\,\alpha : K.\, B)^\div$.

By definition of the translation, we must show that

$\Gamma^+ \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha}_{ans} : \star.\, \boldsymbol{\lambda}\,\mathbf{k} : (\boldsymbol{\Pi}\,\boldsymbol{\alpha} : K^+.\, B^\div) \rightarrow \boldsymbol{\alpha}.$
$\quad (\mathbf{k}\ (\boldsymbol{\lambda}\,\boldsymbol{\alpha} : K^+.\, e^\div)) : \boldsymbol{\Pi}\,\boldsymbol{\alpha}_{ans} : \star.\, (\boldsymbol{\Pi}\,\boldsymbol{\alpha} : K^+.\, B^\div \rightarrow \boldsymbol{\alpha}_{ans}) \rightarrow \boldsymbol{\alpha}_{ans}$

It suffices to show that

$\Gamma^+, \boldsymbol{\alpha}_{ans} : \star, \mathbf{k} : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : K^+.\, B^\div \rightarrow \boldsymbol{\alpha}_{ans} \vdash \mathbf{k}\ (\boldsymbol{\lambda}\,\boldsymbol{\alpha} : K^+.\, e^\div) : \boldsymbol{\alpha}_{ans}.$

By Rule App, it suffices to show that

$\Gamma^+, \boldsymbol{\alpha}_{ans} : \star, \mathbf{k} : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : K^+.\, B^\div \rightarrow \boldsymbol{\alpha}_{ans} \vdash (\boldsymbol{\lambda}\,\mathbf{x} : A^\div.\, e^\div) : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : K^+.\, B^\div$

By part 2 of the induction hypothesis applied to $\Gamma, \alpha : K \vdash e : B$, we know that

$\Gamma^+, \boldsymbol{\alpha} : K^+ \vdash e^\div : B^\div$

It suffices to show that $\vdash \Gamma^+, \boldsymbol{\alpha}_{ans} : \star, \mathbf{k} : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : K^+.\, B^\div \rightarrow \boldsymbol{\alpha}_{ans}$ which follows easily by parts 5 and 4 of the induction hypothesis applied to the typing derivations for K and B.

Case: Rule App

Sub-case: A term-level function applied to a term $\Gamma \vdash e_1\ e_2 : B[e_2/x]$

We must show that

$\Gamma^+ \vdash (e_1\ e_2)^\div : (B[e_2/x])^\div$

By definition of the translation, we must show:

$\Gamma^+ \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\, \boldsymbol{\lambda}\,\mathbf{k} : (B[e_2/x])^+ \rightarrow \boldsymbol{\alpha}.$
$\quad e_1^\div\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : A^\div.\, B^\div.\, (\mathbf{f}\ e_2^\div)\ \boldsymbol{\alpha}\ \mathbf{k}) : (B[e_2/x])^\div$

By part 6 of Lemma 6.5.1 and definition of $B^\div$, we must show:

$$\Gamma^{+} \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\,\boldsymbol{\lambda}\,\mathbf{k} : (\mathsf{B}^{+}[\mathsf{e}_2^{\div}/\mathbf{x}]) \rightarrow \boldsymbol{\alpha}.\ \mathsf{e}_1^{\div}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{\div}.\,\mathsf{B}^{\div}.\ (\mathbf{f}\ \mathsf{e}_2^{\div})\ \boldsymbol{\alpha}\ \mathbf{k}) \quad : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(\mathsf{B}^{+}[\mathsf{e}_2^{\div}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$$

It suffices to show that

- $\Gamma^{+} \vdash \mathsf{B}^{+}[\mathsf{e}_2^{\div}/\mathbf{x}] : \boldsymbol{\kappa}$ By part 3 of the induction hypothesis we know that $\Gamma^{+}, \mathbf{x} : \mathsf{A}^{\div} \vdash \mathsf{B}^{+} : \boldsymbol{\kappa}$, and by part 2 of the induction hypothesis we know that $\Gamma^{+} \vdash \mathsf{e}_2^{\div} : \mathsf{A}^{\div}$, hence the goal follows by substitution.
- $\Gamma^{+} \vdash \mathsf{e}_1^{\div} : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(\boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{\div}.\,\mathsf{B}^{\div} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$, which follows by part 2 of the induction hypothesis and by definition of $(\Pi\,x : A.\,B)^{\div}$.
- $\Gamma^{+}, \boldsymbol{\alpha} : \star, \mathbf{k} : (\mathsf{B}^{+}[\mathsf{e}_2^{\div}/\mathbf{x}]) \rightarrow \boldsymbol{\alpha} \vdash (\boldsymbol{\lambda}\,\mathbf{f} : \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{\div}.\,\mathsf{B}^{\div}.\,(\mathbf{f}\ \mathsf{e}_2^{\div})\ \boldsymbol{\alpha}\ \mathbf{k}) : \boldsymbol{\Pi}\,\mathbf{x} : \mathsf{A}^{\div}.\,\mathsf{B}^{\div} \rightarrow \boldsymbol{\alpha}$, which follows since by part 2 of the induction hypothesis $\mathsf{e}_2^{\div} : \mathsf{A}^{\div}$ we know $(\mathbf{f}\ \mathsf{e}_2^{\div}) : \mathsf{B}^{\div}[\mathsf{e}_2^{\div}/\mathbf{x}]$ and by definition $\mathsf{B}^{\div}[\mathsf{e}_2^{\div}/\mathbf{x}] = \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(\mathsf{B}^{+}[\mathsf{e}_2^{\div}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$.

Sub-case: A term-level function applied to a type $\Gamma \vdash e_1\ A : B[A/\alpha]$

The proof is similar to the previous case, but relies on showing that $\Gamma^{+} \vdash A^{+} : K^{+}$, which follows by part 3 of the induction hypothesis.

Sub-case: A type-level function applied to a term $\Gamma \vdash A\ e : K[e/x]$

This case is straightforward by the part 3 and part 2 of the induction hypothesis.

Sub-case: A type-level function applied to a type $\Gamma \vdash A\ B : K[B/\alpha]$

This case is straightforward by the part 3 of the induction hypothesis.

Case: Rule Conv $\Gamma \vdash e : A$ such that $\Gamma \vdash e : B$ and $A \equiv B$.

We must show that $e^{\div}$ has type $A^{\div} = \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(A^{+} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$.

By the induction hypothesis, we know that $e^{\div} : B^{\div} = \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\,(B^{+} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$. By Rule Conv it suffices to show that $A^{+} \equiv B^{+}$, which follows by Lemma 6.5.4. ☐

To recover a simple statement of the type-preservation theorem over the PTS syntax, I define two meta-functions for translating expressions depending on their use. I define CPS $[\![t]\!]$ to translate a PTS expression in "term" position, *i.e.*, when used on the left side of a type annotation as in $t : t'$, and define CPST $[\![t']\!]$ to translate an expression in "type" position, *i.e.*, when used on the right side of a type annotation. I define these in terms of the translation shown above, noting that for every $t : t'$ in the PTS syntax, one of the following is true: t is a term e and t' is a type A in the explicit syntax; t is a type A and t' is a kind K in the explicit syntax; or t is a kind K and t' is a universe U in the explicit syntax.

$$\begin{aligned} \text{CPS}\,[\![t]\!] &\overset{\text{def}}{=} e^{\div} \text{ when } t \text{ is a term} & \qquad \text{CPST}\,[\![t']\!] &\overset{\text{def}}{=} A^{\div} \text{ when } t' \text{ is a type} \\ \text{CPS}\,[\![t]\!] &\overset{\text{def}}{=} A^{+} \text{ when } t \text{ is a type} & \qquad \text{CPST}\,[\![t']\!] &\overset{\text{def}}{=} K^{+} \text{ when } t' \text{ is a kind} \\ \text{CPS}\,[\![t]\!] &\overset{\text{def}}{=} K^{+} \text{ when } t \text{ is a kind} & \qquad \text{CPST}\,[\![t']\!] &\overset{\text{def}}{=} U^{+} \text{ when } t' \text{ is a universe} \end{aligned}$$

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

$$\frac{\Gamma \vdash \mathbf{e} : \Pi\, \alpha : \mathbf{Prop}\,.\,(\mathbf{bool} \rightarrow \alpha) \rightarrow \alpha}{\Gamma \vdash \mathbf{e}} \qquad \frac{\cdot \vdash \mathbf{e}}{\vdash \mathbf{e}}$$

Figure 6.14: CoC^k Components and Programs

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

$$\mathbf{eval}(\mathbf{e}) \quad = \quad \mathbf{v} \quad \text{if} \vdash \mathbf{e} \text{ and } \cdot \vdash \mathbf{e}\ @\ \mathbf{bool}\ \mathbf{id} \triangleright^* \mathbf{v}$$

Figure 6.15: CoC^k Evaluation

This notation is based on Barthe and Uustalu (2002).

Theorem 6.5.6 (CPS^n Type Preservation). $\Gamma \vdash t : t'$ *then* $\Gamma^+ \vdash \text{CPS}\,[\![t]\!] : \text{CPST}\,[\![t']\!]$

6.5.2 Compiler Correctness

Recall from Chapter 3 that since we preserve conversion, proving compiler correctness is simple. I use the standard definition of linking by substitution and the standard cross-language relation.

I extend the CPS^n translation in a straightforward way to translate closing substitutions, written $\gamma^{\div}$, and allow translated terms to be linked with the translation of any valid closing substitution γ. This definition supports a separate compilation theorem that allows linking with the output of this translation, but not with the output of other compilers.

Now I can show that the CPS^n translation is correct with respect to separate compilation—if we first link and run to a value, we get a related value when we compile and then link with the compiled closing substitution. I first define well-formed programs and the evaluation function for CoC^k. These are different than described in Chapter 2. I present well-formed programs and components in Figure 6.14. CPS programs are well-formed when they are computations that produce a ground values, *i.e.*, when they expect a continuation that expects a value of type $\mathbf{bool}$. The evaluation function is given in Figure 6.15 Since the target language is in CPS, we must apply the halt continuation $\mathbf{id}$ at the top-level to evaluate a program to an observation.

Theorem 6.5.7 (Separate Compilation Correctness). *If* $\Gamma \vdash e$ *and* $\Gamma \vdash \gamma$*, then* $\text{eval}(\gamma(e)) \approx \mathbf{eval}(\gamma^+(e^{\div}))$.

Proof. Since conversion implies equivalence, we reason in terms of equivalence. By Lemma 6.5.3, $(\gamma(e))^{\div} \triangleright^* \mathbf{e}$ and $v^{\div} \equiv \mathbf{e}$. By Lemma 6.5.1, $(\gamma(e))^{\div} \equiv \gamma^{\div}(e^{\div})$, hence $\gamma^{\div}(e^{\div}) \triangleright^* \mathbf{e}$ and $v^{\div} \equiv \mathbf{e}$. Since the translation on all observations is $v^{\div} = \boldsymbol{\lambda\, \alpha.\, \lambda\, k.\, k\ v}$, where $v \approx \mathbf{v}$, we know $v^{\div}\ A^+\ \mathbf{id} \triangleright^* \mathbf{v}$ such that $v \approx \mathbf{v}$. Since $v^{\div} \equiv \mathbf{e} \equiv \gamma^{\div}(e^{\div})$, we

$\boxed{\Gamma \vdash K : U \leadsto^{v}_{\kappa} \boldsymbol{\kappa} \text{ Lemma 6.6.6 will show } \Gamma^{+} \vdash K^{+} : U^{+}}$

$$\frac{}{\Gamma \vdash \star : \square \leadsto^{v}_{\kappa} \boldsymbol{\star}}\ \text{CPS}^{v}_{\kappa}\text{-Ax} \qquad \frac{\Gamma \vdash K : U \leadsto^{v}_{\kappa} \boldsymbol{\kappa} \qquad \Gamma, \alpha : K \vdash K' : U \leadsto^{v}_{\kappa} \boldsymbol{\kappa}'}{\Gamma \vdash \Pi\, \alpha : K.\, K' : U \leadsto^{v}_{\kappa} \boldsymbol{\Pi\, \alpha : \kappa.\, \kappa'}}\ \text{CPS}^{v}_{\kappa}\text{-PIK}$$

$$\frac{\Gamma \vdash A : K' \leadsto^{v}_{A} \mathbf{A} \qquad \Gamma, x : A \vdash K : U \leadsto^{v}_{\kappa} \boldsymbol{\kappa}}{\Gamma \vdash \Pi x : A.\, K : U \leadsto^{v}_{\kappa} \boldsymbol{\Pi\, x : A.\, \kappa}}\ \text{CPS}^{v}_{\kappa}\text{-PIA}$$

Figure 6.16: CPS^{v} of Kinds

also know that $\gamma^{\div}(e^{\div})\ A^{+}\mathbf{id} \triangleright^{*} \mathbf{v}'$ and $\mathbf{v}' \equiv \mathbf{v}$. Since $\mathbf{v}$ is an observation, $\mathbf{v}' = \mathbf{v}$ and $v \approx \mathbf{v}'$. □

Corollary 6.5.8 (Whole-Program Correctness)**.** *If* $\vdash e$ *then* $\mathsf{eval}(e) \approx \mathbf{eval}(e^{\div})$.

6.6 Call-By-Value CPS Translation

In this section, I present the call-by-value (CBV) CPS translation (CPS^{v}) of CoC^{D}. First, I redefine the short-hand from Section 6.5 to refer to call-by-value translation.

$$A^{\div} \overset{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash A : \star \leadsto^{v}_{A^{\div}} \mathbf{A} \qquad U^{+} \overset{\text{def}}{=} \mathbf{U} \text{ where } \Gamma \vdash U \leadsto^{v}_{U} \mathbf{U}$$

$$e^{\div} \overset{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash e : A \leadsto^{v}_{e} \mathbf{e} \qquad K^{+} \overset{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash K : U \leadsto^{v}_{\kappa} \boldsymbol{\kappa}$$

$$A^{+} \overset{\text{def}}{=} \mathbf{A} \text{ where } \Gamma \vdash A : K \leadsto^{v}_{A} \mathbf{A}$$

In general, CPS^{v} differs from CPS^{n} in two ways. First, all term variables must have value types, so the translation rules for all binding constructs now use the value translation for type annotations. Second, we change the definition of value types so that functions must receive values are arguments and pairs must contain values as components. Since the translation must force every computation to a value, the translation of every feature of dependency requires the new typing rule Rule T-Cont. Furthermore, all substitutions of a term into a type must substitute *values* instead of computations, so all dependent type annotations must explicitly convert computations to values by supplying the identity continuation.

The translation of universes is unchanged compared to the CBN translation, so I leave its definition in Appendix I Figure I.2.

I define the translation of kinds in Figure 6.16. The only difference is in Rule CPS$^{v}_{\kappa}$-PiA. Now Rule CPS$^{v}_{\kappa}$-PiA translates the kind of type-level functions $\Pi x : A.\, K$ to accept a *value* as argument $\mathbf{x} : A^{+}$. In the CBN translation, the domain of a dependent function is a computation, so the domain annotation is translated with the computation translation on types. Now, in the CBV translation, all arguments to dependent functions

$$\boxed{\Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{A} \text{ Lemma 6.6.6 will show } \Gamma^{+} \vdash \mathsf{A}^{+} : \mathsf{K}^{+}}$$

$$\dfrac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A^{\div}} \mathbf{B}}{\Gamma \vdash \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{\Pi\, x : A.\, B}} \ \text{CPS}^{v}_{A}\text{-}\textsc{Pi}$$

$$\dfrac{\Gamma \vdash \mathsf{K} : \mathsf{U}' \rightsquigarrow^{v}_{\kappa} \boldsymbol{\kappa} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{U} \rightsquigarrow^{v}_{A^{\div}} \mathbf{B}}{\Gamma \vdash \Pi\, \alpha : \mathsf{K}.\, \mathsf{B} : \mathsf{U} \rightsquigarrow^{v}_{A} \boldsymbol{\Pi\, \alpha : \kappa.\, B}} \ \text{CPS}^{v}_{A}\text{-}\textsc{PiK}$$

$$\dfrac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{B}}{\Gamma \vdash \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} : \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{K} \rightsquigarrow^{v}_{A} \boldsymbol{\lambda\, x : A.\, B}} \ \text{CPS}^{v}_{A}\text{-}\textsc{Constr}$$

$$\dfrac{\begin{matrix}\Gamma \vdash \mathsf{A} : \Pi\, \mathsf{x} : \mathsf{B}.\, \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{A} \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{B} \qquad \Gamma \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e}\end{matrix}}{\Gamma \vdash \mathsf{A}\ \mathsf{e} : \mathsf{K}[\mathsf{e}/\mathsf{x}] \rightsquigarrow^{v}_{A} \mathbf{A}\ (\mathbf{e}\ \mathbf{B}\ \mathbf{id})} \ \text{CPS}^{v}_{A}\text{-}\textsc{AppConstr}$$

$$\dfrac{\Gamma \vdash \mathsf{A} : \star \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^{v}_{A} \mathbf{B}}{\Gamma \vdash \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} : \star \rightsquigarrow^{v}_{A} \mathbf{\Sigma\, x : A.\, B}} \ \text{CPS}^{v}_{A}\text{-}\textsc{Sigma}$$

$$\dfrac{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^{v}_{e} \mathbf{e} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{B}}{\Gamma \vdash \mathsf{let}\, \mathsf{x} = \mathsf{e} : \mathsf{A}\, \mathsf{in}\, \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{let\, x = e\ A\ id : A\, in\, B}} \ \text{CPS}^{v}_{A}\text{-}\textsc{Let} \qquad \cdots$$

Figure 6.17: CPS^{v} of Types (excerpts)

are values, so Rule CPS$^{v}_{\kappa}$-PiA uses the value translation on types to translate the domain annotation.

I present the translation on types in Figure 6.17. The type translation has multiple rules with type annotations that have changed from CBN. The computation translation of types is unchanged. In the value translation of types, similar to the kind translation, dependent function types that abstract over terms now translate the domain annotation $\mathsf{x} : \mathsf{A}$ using the value translation. After CBV translation, dependent pairs must contain values, so the translation of $\Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B}$ uses the value translation on the component types, *i.e.*, the CBV translation is $\mathbf{\Sigma\, x} : \mathsf{A}^{+}.\, \mathsf{B}^{+}$. When terms appear in the type language, such as in CPS$^{v}_{A}$-AppConstr and CPS$^{v}_{A}$-Let, we must explicitly convert the computation to a value to maintain the invariant that all term variables refer to values. For example, in CPS$^{v}_{A}$-AppConstr we translate a type-level application with a term argument $\mathsf{A}\ \mathsf{e}$ to $\mathsf{A}^{+}\ (\mathsf{e}^{\div}\ \mathsf{B}^{+}\ \mathbf{id})$. We similarly translate let-bound terms CPS$^{v}_{A}$-Let by casting the computation to a value. Recall from Section 6.2 that, while expressions of the form $\mathsf{A}^{+}\ (\mathsf{e}^{\div}\ \mathsf{B}^{+}\ \mathbf{id})$ are not in CPS, this expression is a type and will be evaluated during type checking. Terms that evaluate at run time are always in CPS and never return.

The term translation (Figure 6.18 and Figure 6.19) changes in three major ways. As in Section 6.5, we implicitly have a computation and a value translation on term values, with the latter inlined into the former. First, unlike in CPS^{n}, variables are

$\boxed{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^{v}_{e} \mathbf{e} \text{ Lemma 6.6.6 will show } \Gamma^{+} \vdash \mathsf{e}^{\div} : \mathsf{A}^{\div}}$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{A}}{\Gamma \vdash \mathsf{x} : \mathsf{A} \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : A \to \alpha. k\ x}}\ \textsc{CPS}^{v}_{e}\text{-}\textsc{Var}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A\div} \mathbf{B} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e}}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}. \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}. \mathsf{B} \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : (\Pi x : A. B) \to \alpha. k\ (\lambda x : A. e)}}\ \textsc{CPS}^{v}_{e}\text{-}\textsc{Fun}$$

$$\frac{\Gamma \vdash \mathsf{K} : \_ \rightsquigarrow^{v}_{\kappa} \boldsymbol{\kappa} \qquad \Gamma, \alpha : \mathsf{K} \vdash \mathsf{B} : \_ \rightsquigarrow^{v}_{A\div} \mathbf{B} \qquad \Gamma, \alpha : \mathsf{K} \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e}}{\begin{array}{c}\Gamma \vdash \lambda \alpha : \mathsf{K}. \mathsf{e} : \Pi \alpha : \mathsf{K}. \mathsf{B} \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha_{ans} : \star. \lambda k : (\Pi \alpha : \kappa. B) \to \alpha_{ans}.} \\ \boldsymbol{k\ (\lambda \alpha : \kappa. e)}\end{array}}\ \textsc{CPS}^{v}_{e}\text{-}\textsc{Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \Pi \mathsf{x} : \mathsf{A}. \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A\div} \mathbf{B}^{\div} \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{B}^{+} \qquad \Gamma \vdash \mathsf{e}' : \mathsf{A} \rightsquigarrow^{v}_{e} \mathbf{e}' \qquad \Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{A}\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{e}\ \mathsf{e}' : \mathsf{B}[\mathsf{e}'/\mathsf{x}] \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : (B^{+}[(e'\ A\ id)/x]) \to \alpha.} \\ \boldsymbol{e\ \alpha\ (\lambda f : \Pi x : A. B^{\div}.} \\ \boldsymbol{e'\ @\ \alpha\ (\lambda x : A. (f\ x)\ \alpha\ k))}\end{array}}\ \textsc{CPS}^{v}_{e}\text{-}\textsc{App}$$

$$\frac{\Gamma \vdash \mathsf{e} : \Pi \alpha : \mathsf{K}. \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e} \qquad \Gamma, \alpha : \mathsf{K} \vdash \mathsf{B} : \_ \rightsquigarrow^{v}_{A\div} \mathbf{B} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^{v}_{e} \mathbf{A}}{\begin{array}{c}\Gamma \vdash \mathsf{e}\ \mathsf{A} : \mathsf{let}\, \mathsf{x} = \mathsf{A}\, \mathsf{in}\, \mathsf{B} \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha_{ans} : \star. \lambda k : (B[A/\alpha]) \to \alpha_{ans}.} \\ \boldsymbol{e\ \alpha\ (\lambda f : \Pi \alpha : \kappa. B.} \\ \boldsymbol{(f\ A)\ \alpha_{ans}\ k)}\end{array}}\ \textsc{CPS}^{v}_{e}\text{-}\textsc{Inst}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \rightsquigarrow^{n}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : bool \to \alpha. k\ true}}\ \textsc{CPS}^{n}_{e}\text{-}\textsc{True}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \rightsquigarrow^{n}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : bool \to \alpha. k\ false}}\ \textsc{CPS}^{n}_{e}\text{-}\textsc{False}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{bool} \rightsquigarrow^{v}_{e} \mathbf{e} \\ \Gamma \vdash \mathsf{B} : \star \rightsquigarrow^{v}_{A} \mathbf{B} \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e}_2\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{if}\, \mathsf{e}\, \mathsf{then}\, \mathsf{e}_1\, \mathsf{else}\, \mathsf{e}_2 : \mathsf{B} \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : B \to \alpha.} \\ \boldsymbol{e\ @\ \alpha\ (\lambda x : bool.\, if\, x\, then\, (e_1\ @\ \alpha\ k)} \\ \boldsymbol{else\, (e_2\ @\ \alpha\ k))}\end{array}}\ \textsc{CPS}^{n}_{e}\text{-}\textsc{If}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^{v}_{e} \mathbf{e} \\ \Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^{v}_{A} \mathbf{A} \qquad \Gamma \vdash \mathsf{B} : \mathsf{K} \rightsquigarrow^{v}_{A} \mathbf{B} \qquad \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A} \vdash \mathsf{e}' : \mathsf{B} \rightsquigarrow^{v}_{e} \mathbf{e}'\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{let}\, \mathsf{x} = \mathsf{e} : \mathsf{A}\, \mathsf{in}\, \mathsf{e}' : \mathsf{B}[\mathsf{e}/\mathsf{x}] \rightsquigarrow^{v}_{e} \boldsymbol{\lambda \alpha : \star. \lambda k : B[(e\ A\ id)/x] \to \alpha.} \\ \boldsymbol{e\ @\ \alpha\ (\lambda x : A. e'\ \alpha\ k)}\end{array}}\ \textsc{CPS}^{v}_{e}\text{-}\textsc{Let}$$

Figure 6.18: CPS^{v} of Terms (1/2)

$$\dfrac{\begin{array}{c}\Gamma \vdash e_1 : A \leadsto^v_e \mathbf{e_1} \\ \Gamma \vdash e_2 : B[e_1/x] \leadsto^v_e \mathbf{e_2} \qquad \Gamma \vdash A : \star \leadsto^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \leadsto^v_A \mathbf{B}\end{array}}{\begin{array}{l}\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma\, x : A.\, B \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : \Sigma\, x : A.\, B \to \alpha.} \\ \qquad \mathbf{e_1} \ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x_1 : A.} \\ \qquad\qquad \mathbf{e_2}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x_2 : B[(e_1\ A\ id)/x].} \\ \qquad\qquad\qquad \mathbf{k}\ \langle \mathbf{x_1}, \mathbf{x_2} \rangle\ \mathbf{as}\ \boldsymbol{\Sigma\, x : A.\, B}))\end{array}}\ \text{CPS}^v_e\text{-Pair}$$

$$\dfrac{\Gamma \vdash A : \star \leadsto^v_A \mathbf{A} \qquad \Gamma \vdash e : \Sigma\, x : A.\, B \leadsto^v_e \mathbf{e}}{\begin{array}{l}\Gamma \vdash \mathsf{fst}\, e : A \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : A^+ \to \alpha.} \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, y : \Sigma\, x : A.\, B.}\ \mathbf{let}\ \mathbf{z} = \mathbf{fst}\ \mathbf{y}\ \mathbf{in}\ \mathbf{k}\ \mathbf{z})\end{array}}\ \text{CPS}^v_e\text{-Fst}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash A : \star \leadsto^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \leadsto^v_A \mathbf{B} \\ \Gamma \vdash (\mathsf{fst}\, e) : A \leadsto^v_e (\mathsf{fst}\, e)^{\div} \qquad \Gamma \vdash e : \Sigma\, x : A.\, B \leadsto^v_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x] \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : B[((\mathsf{fst}\, e)^{\div}\ A\ id)/x] \to \alpha.} \\ \qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, y : \Sigma\, x : A.\, B.}\ \mathbf{let}\ \mathbf{z} = \mathbf{snd}\ \mathbf{y}\ \mathbf{in}\ \mathbf{k}\ \mathbf{z})\end{array}}\ \text{CPS}^v_e\text{-Snd}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash e_1 : A \leadsto^v_e \mathbf{e_1} \\ \Gamma \vdash e_2 : B[e_1/x] \leadsto^v_e \mathbf{e_2} \qquad \Gamma \vdash A : \star \leadsto^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \leadsto^v_A \mathbf{B}\end{array}}{\begin{array}{l}\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma\, x : A.\, B \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : \Sigma\, x : A.\, B \to \alpha.} \\ \qquad \mathbf{e_1}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x : A.} \\ \qquad\qquad \mathbf{e_2}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x_2 : B.}\ \mathbf{k}\ \langle \mathbf{x}, \mathbf{x_2} \rangle))\end{array}}\ \text{CPS}^v_e\text{-Pair-Alt}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash A : K \leadsto^v_A \mathbf{A} \qquad \Gamma \vdash K : U \leadsto^v_\kappa \boldsymbol{\kappa} \\ \Gamma, \alpha = A : K \vdash B : K' \leadsto^v_A \mathbf{B} \qquad \Gamma, \alpha = A : K \vdash e : B \leadsto^v_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\, \alpha = A : K\, \mathsf{in}\, e : B[A/\alpha] \leadsto^v_e \boldsymbol{\lambda\, \alpha_{ans} : \star.\, \lambda\, k : B[A/\alpha] \to \alpha_{ans}.} \\ \qquad \mathbf{let}\ \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}\ \mathbf{in}\ \mathbf{e}\ \boldsymbol{\alpha_{ans}}\ \mathbf{k}\end{array}}\ \text{CPS}^v_e\text{-LetK}$$

$$\dfrac{\Gamma \vdash e : B \leadsto^v_e \mathbf{e}}{\Gamma \vdash e : A \leadsto^v_e \mathbf{e}}\ \text{CPS}^v_e\text{-Conv}$$

Figure 6.19: CPS^v of Terms (2/2)

values, whereas the translation must produce a computation. Therefore, we translate x by "value η-expansion" into $\boldsymbol{\lambda\, \alpha.\, \lambda\, k.\, k\ x}$, a computation that immediately applies the continuation to the value. Second, as discussed above, we change the translation of application CPS^v_e-App to force the evaluation of the function argument. Third, in the translation of pairs CPS^v_e-Pair, we force the evaluation of the components of the pair and produce a pair of values for the continuation. Note that in cases of the translation where we have types with dependency—CPS^v_e-App, CPS^v_e-Let, CPS^v_e-Pair, and CPS^v_e-Snd—we cast computations to values in the types by applying the identity continuation, and require the @ form to use the new typing rule Rule T-Cont.

$\boxed{\vdash \Gamma \leadsto^{v} \boldsymbol{\Gamma} \text{ Lemma 6.6.6 will show } \vdash \Gamma^{+}}$

$$\frac{}{\vdash \cdot \leadsto^{v} \cdot}\ \text{CPS}^{v}_{\Gamma}\text{-Empty} \qquad \frac{\vdash \Gamma \leadsto^{v} \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^{v}_{A} \mathbf{A}}{\vdash \Gamma, \mathsf{x} : \mathsf{A} \leadsto^{v} \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A}}\ \text{CPS}^{v}_{\Gamma}\text{-AssumT}$$

$$\frac{\vdash \Gamma \leadsto^{v} \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^{v}_{\kappa} \boldsymbol{\kappa}}{\vdash \Gamma, \alpha : \mathsf{K} \leadsto^{v} \boldsymbol{\Gamma}, \boldsymbol{\alpha} : \boldsymbol{\kappa}}\ \text{CPS}^{v}_{\Gamma}\text{-AssumK}$$

$$\frac{\vdash \Gamma \leadsto^{v} \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^{v}_{A} \mathbf{A} \qquad \Gamma \vdash \mathsf{e} : \mathsf{A} \leadsto^{v}_{e} \mathbf{e}}{\vdash \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A} \leadsto^{v} \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e}\ \mathbf{A}\ \mathbf{id} : \mathbf{A}}\ \text{CPS}^{v}_{\Gamma}\text{-Def}$$

$$\frac{\vdash \Gamma \leadsto^{v} \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \leadsto^{v}_{A} \mathbf{A} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^{v}_{\kappa} \boldsymbol{\kappa}}{\vdash \Gamma, \alpha = \mathsf{A} : \mathsf{K} \leadsto^{v} \boldsymbol{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}}\ \text{CPS}^{v}_{\Gamma}\text{-DefT}$$

Figure 6.20: CPS^{v} of Environments

DIGRESSION Interestingly, because typing the application of a continuation is essentially a dependent let, we can simplify the translation of pairs. I present this in the rule CPS$^{v}_{e}$-Pair-Alt. Instead of explicitly substituting the value of $\mathbf{e}_1$ into the type $\mathbf{B}$, we simply use the same variable name $\mathbf{x}$ to bind the value of $\mathbf{e}_1$ in both the term and the type. Since that variable is free in the type annotation $\mathbf{B}$ on the variable $\mathbf{x}_2$, we implicitly substitute its value into $\mathbf{B}$ rather than being so explicitly. This is rather subtle so I prefer the more direct and explicit translation, CPS$^{v}_{e}$-Pair.

Given the translation of binding constructs in the language, the translation of the typing environment (Figure 6.20) should be unsurprising. Since all variables are values, we translate term variables $\mathsf{x} : \mathsf{A}$ using the value translation on types to produce $\mathbf{x} : \mathsf{A}^{+}$ instead of $\mathbf{x} : \mathsf{A}^{\div}$. We must also translate term definitions $\mathsf{x} = \mathsf{e} : \mathsf{A}$ by casting the computation to a value, producing $\mathbf{x} = \mathsf{e}^{\div}\ \mathsf{A}^{+}\ \mathbf{id} : \mathsf{A}^{+}$.

6.6.1 Type Preservation

The type-preservation proof follows the standard architecture presented in Chapter 3. First we prove compositionality, then preservation of reduction, preservation of conversion, then preservation of equivalence, then preservation of subtyping, then type preservation. However, we must reason about CPS'd "values" of the form $\mathbf{e}\ @\ \mathbf{A}\ \mathbf{k}$ in all uses of dependency. This requires a few extra steps of reasoning, particularly in the proof of Lemma 6.6.2 (CPS^{v} Compositionality). Otherwise, the proofs of the supporting lemmas are essentially the same as in Section 6.5.

I begin with a technical lemma that is essentially an η-principle for CPS'd computations, which simplifies the aforementioned reasoning about CPS'd "values".[3] The lemma states that any CPS'd computation $e^{\div}$ is equivalent to a new CPS'd computation that accepts a continuation $\mathbf{k}$ simply applies $e^{\div}$ to $\mathbf{k}$. The proof is straightforward. Note the type annotations are apparently mismatched, as in our explanation of the translation of Rule Conv in Section 6.5 and the discussion of untyped vs type equivalence in Chapter 3, but the behaviors of the terms are the same and equivalence is untyped.

Lemma 6.6.1 (CPS^v Computation η). $e^{\div} \equiv \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.}\, e^{\div}\ \boldsymbol{@\ \alpha\ (\lambda\,x : B.\,k\ x)}$

Proof. Note that $e^{\div} \equiv \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.}\, e^{\div}\ \boldsymbol{\alpha\ (\lambda\,x : B.\,k\ x)}$, by η-equivalence. By transitivity, it suffices to show that
$\boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.}\, e^{\div}\ \boldsymbol{\alpha\ (\lambda\,x : B.\,k\ x)} \equiv \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.}\, e^{\div}\ \boldsymbol{@\ \alpha\ (\lambda\,x : B.\,k\ x)}$

Intuitively, this is true since $\boldsymbol{@}$ dynamically behaves exactly like application, only changing which typing rule is used. Since equivalence is untyped, the semantics of $\boldsymbol{@}$ as far as equivalence is concerned is no different than normal application.

Note that by definition of the translation, $e^{\div}$ must be of the form $\boldsymbol{\lambda\,\alpha.\,\lambda\,k.\,e'}$.

The goal follows since
$\boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.\,(\lambda\,\alpha.\,\lambda\,k.\,e')\ \alpha\ (\lambda\,x : B.\,k\ x)} \triangleright^{*} \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.\,e'}$
and
$\boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.\,(\lambda\,\alpha.\,\lambda\,k.\,e')\ @\ \alpha\ (\lambda\,x : B.\,k\ x)} \triangleright^{*} \boldsymbol{\lambda\,\alpha : \star.\,\lambda\,k : A \rightarrow \alpha.\,e'}$ ☐

Since variables are values in call-by-value, we adjust the statement of Lemma 6.6.2 to cast computations to values. Proving this lemma now requires the new equivalence rule Rule ≡-Cont for cases involving substitution of terms. Recall that all terms being translated have an implicit typing derivation, so the omitted types are easy to reconstruct.

Lemma 6.6.2 (CPS^v Compositionality).

1. $(K[A/\alpha])^{+} \equiv K^{+}[A^{+}/\boldsymbol{\alpha}]$
2. $(K[e/x])^{+} \equiv K^{+}[e^{\div}\ \_\ \mathbf{id}/\mathbf{x}]$
3. $(A[B/\alpha])^{+} \equiv A^{+}[B^{+}/\boldsymbol{\alpha}]$
4. $(A[e/x])^{+} \equiv A^{+}[e^{\div}\ \_\ \mathbf{id}/\mathbf{x}]$
5. $(A[B/\alpha])^{\div} \equiv A^{\div}[B^{+}/\boldsymbol{\alpha}]$
6. $(A[e/x])^{\div} \equiv A^{\div}[e^{\div}\ \_\ \mathbf{id}/\mathbf{x}]$
7. $(e[A/\alpha])^{\div} \equiv e^{\div}[A^{+}/\boldsymbol{\alpha}]$
8. $(e[e'/x])^{\div} \equiv e^{\div}[e'^{\div}\ \_\ \mathbf{id}/\mathbf{x}]$

Proof. The proof is straightforward by induction on the typing derivation of the expression t being substituted into.

Part 6 follows immediately by part 3 of the induction hypothesis. Part 7 follows immediately by part 4 of the induction hypothesis. I give representative cases for the other parts. In most cases, it suffices to show that the two terms are syntactically identical.

3 The proofs for the CBN setting only require a specialized instance of this property although the general form holds.

Case: Rule * $t = U$, parts 1 and 2. Trivial, since no free variables appear in U.

Case: Rule Pi-* $t = \Pi x : B.\, K'$

Sub-case: Part 1. We must show that $((\Pi x : B.\, K')[A/\alpha])^+ = (\Pi x : B.\, K')^+[A^+/\boldsymbol{\alpha}]$.

$$
\begin{aligned}
& ((\Pi x : B.\, K')[A/\alpha])^+ \\
= \; & (\Pi x : B[A/\alpha].\, K'[A/\alpha])^+ && (115)\\
& \text{by definition of substitution} \\
= \; & \boldsymbol{\Pi}\, \mathbf{x}' : (B[A/\alpha])^+.\, (K'[A/\alpha])^+ && (116)\\
& \text{by definition of the translation} \\
= \; & \boldsymbol{\Pi}\, \mathbf{x}' : B^+[A^+/\alpha].\, K'^+[A^+/\alpha] && (117)\\
& \text{by parts 1 and 3 of the induction hypothesis} \\
= \; & (\boldsymbol{\Pi}\, \mathbf{x}' : B^+.\, K'^+)[A^+/\boldsymbol{\alpha}] && (118)\\
& \text{by definition of substitution} \\
= \; & (\Pi x' : B.\, K')^+[A^+/\boldsymbol{\alpha}] && (119)\\
& \text{by definition of the translation}
\end{aligned}
$$

Sub-case: Part 2. Similar to the previous subcase.

Case: Rule Var

Sub-case: $t = \alpha'$, part 3. Part 4 is trivial since x is not free in α.

We must show that $(\alpha'[A/\alpha])^+ = \boldsymbol{\alpha}'[A^+/\boldsymbol{\alpha}]$.

Sub-case: $\alpha = \alpha'$. It suffices to show that $A^+ = A^+$, which is trivial.

Sub-case: $\alpha \neq \alpha'$. Trivial.

Sub-case: $t = x$, part 7 is trivial.

Sub-case: Part 8

Case: Rule Var Part 8, $(x[e/x'])^{\div}$

We must show $(x[e/x'])^{\div} = x^{\div}[e^{\div}\ \_\ \mathbf{id}/\mathbf{x}']$.

W.l.o.g., assume $x = x'$.

$$
\begin{aligned}
& (x[e/x])^{\div} \\
= \; & e^{\div} && \text{by definition of substitution} && (120)\\
\equiv \; & \boldsymbol{\lambda}\boldsymbol{\alpha}.\, \boldsymbol{\lambda}\mathbf{k}.\, (e^{\div}\ @\ \boldsymbol{\alpha}\ \boldsymbol{\lambda}\mathbf{x}.\, \mathbf{k}\ \mathbf{x}) && \text{by Lemma 6.6.1} && (121)\\
\equiv \; & \boldsymbol{\lambda}\boldsymbol{\alpha}.\, \boldsymbol{\lambda}\mathbf{k}.\, (\boldsymbol{\lambda}\mathbf{x}.\, \mathbf{k}\ \mathbf{x})\ (e^{\div}\ \_\ \mathbf{id}) && \text{by Rule} \equiv\text{-Cont} && (122)\\
= \; & (\boldsymbol{\lambda}\boldsymbol{\alpha}.\, \boldsymbol{\lambda}\mathbf{k}.\, (\boldsymbol{\lambda}\mathbf{x}.\, \mathbf{k}\ \mathbf{x})\ \mathbf{x})[(e^{\div}\ \_\ \mathbf{id})/\mathbf{x}] && \text{by substitution} && (123)\\
\equiv \; & (\boldsymbol{\lambda}\boldsymbol{\alpha}.\, \boldsymbol{\lambda}\mathbf{k}.\, \mathbf{k}\ \mathbf{x})[(e^{\div}\ \_\ \mathbf{id})/\mathbf{x}] && \text{by } \rhd_\beta && (124)
\end{aligned}
$$

$$= \mathsf{x}^{\div}[(\mathsf{e}^{\div}\ \_\ \mathbf{id})/\mathbf{x}] \qquad \text{by definition of translation} \qquad (125)$$

Case: Rule App $\mathsf{e}_1\ \mathsf{e}_2$

Sub-case: Part 7

We must show $((\mathsf{e}_1\ \mathsf{e}_2)[\mathsf{A}'/\alpha'])^{\div} = (\mathsf{e}_1\ \mathsf{e}_2)^{\div}[\mathsf{A}'^{+}/\boldsymbol{\alpha}']$.

$$((\mathsf{e}_1\ \mathsf{e}_2)[\mathsf{A}'/\alpha'])^{\div}$$

$$= (\mathsf{e}_1[\mathsf{A}/\alpha']\ \mathsf{e}_2[\mathsf{A}'/\alpha'])^{\div} \qquad \text{by substitution} \qquad (126)$$

$$= \boldsymbol{\lambda\,\alpha:\star.\,\lambda\,k}:((\mathsf{B}[\mathsf{A}'/\alpha'])^{+}[(\mathsf{e}_2[\mathsf{A}'/\alpha'])^{\div}/\mathbf{x}]) \to \boldsymbol{\alpha}.$$
$$(\mathsf{e}_1[\mathsf{A}'/\alpha'])^{\div}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f:\Pi\,x}:(\mathsf{A}[\mathsf{A}'/\alpha'])^{+}.\,(\mathsf{B}[\mathsf{A}'/\alpha'])^{\div}.$$
$$(\mathsf{e}_2[\mathsf{A}'/\alpha'])^{\div}\ \mathbf{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,x}:(\mathsf{A}[\mathsf{A}'/\alpha'])^{+}.\,(\mathbf{f\ x})\ \boldsymbol{\alpha}\ \mathbf{k}))$$

$$\text{by translation} \qquad (127)$$

$$= \boldsymbol{\lambda\,\alpha:\star.\,\lambda\,k}:(\mathsf{B}^{+}[\mathsf{A}'^{+}/\boldsymbol{\alpha}'][\mathsf{e}_2^{\div}[\mathsf{A}'^{+}/\boldsymbol{\alpha}']/\mathbf{x}]) \to \boldsymbol{\alpha}.$$
$$\mathsf{e}_1^{\div}[\mathsf{A}'^{+}/\boldsymbol{\alpha}']\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f:\Pi\,x}:\mathsf{A}^{+}[\mathsf{A}'^{+}/\alpha'].\,\mathsf{B}^{\div}[\mathsf{A}'^{+}/\boldsymbol{\alpha}'].$$
$$\mathsf{e}_2^{\div}[\mathsf{A}'^{+}/\boldsymbol{\alpha}']\ \mathbf{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,x}:\mathsf{A}^{+}[\mathsf{A}'^{+}/\boldsymbol{\alpha}'].\,(\mathbf{f\ x})\ \boldsymbol{\alpha}\ \mathbf{k}))$$

$$\text{by IH (3,5,7)} \qquad (128)$$

$$= (\boldsymbol{\lambda\,\alpha:\star.\,\lambda\,k}:(\mathsf{B}^{+}[\mathsf{e}_2^{\div}/\mathbf{x}]) \to \boldsymbol{\alpha}. \qquad (129)$$
$$\mathsf{e}_1^{\div}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,f:\Pi\,x}:\mathsf{A}^{\div}.\,\mathsf{B}^{\div}.$$
$$\mathsf{e}_2^{\div}\ \mathbf{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\,x}:\mathsf{A}^{+}.\,(\mathbf{f\ x})\ \boldsymbol{\alpha}\ \mathbf{k})))[\mathsf{A}'^{+}/\boldsymbol{\alpha}']$$

$$\text{by substitution}$$

$$= (\mathsf{e}_1\ \mathsf{e}_2)^{\div}[\mathsf{A}'^{+}/\boldsymbol{\alpha}'] \qquad \text{by translation} \qquad (130)$$

□

Lemma 6.6.3 (CPS^{v} Preservation of Reduction)**.**

- *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *and* $\mathsf{e} \rhd \mathsf{e}'$ *then* $\mathsf{e}^{\div} \rhd^{*} \mathbf{e}'$ *and* $\mathbf{e}' \equiv \mathsf{e}'^{\div}$
- *If* $\Gamma \vdash \mathsf{A} : \mathsf{K}$ *and* $\mathsf{A} \rhd \mathsf{A}'$ *then* $\mathsf{A}^{+} \rhd^{*} \mathbf{A}'$ *and* $\mathbf{A}' \equiv \mathsf{A}'^{+}$
- *If* $\Gamma \vdash \mathsf{A} : \star$ *and* $\mathsf{A} \rhd \mathsf{A}'$ *then* $\mathsf{A}^{\div} \rhd^{*} \mathbf{A}'$ *and* $\mathbf{A}' \equiv \mathsf{A}'^{\div}$

Proof. The proof is straightforward by cases on the reduction ($\rhd$) relation. I give some representative cases.

Case: $\mathsf{x} \rhd_{\delta} \mathsf{e}'$ where $\mathsf{x} = \mathsf{e}' : \mathsf{A}' \in \Gamma$

We must show that $\mathsf{x}^{\div} \rhd_{\delta} \mathbf{e}$ such that $\mathbf{e} \equiv \mathsf{e}'^{\div}\ \_\ \mathbf{id}$ where $\mathsf{x}^{\div} = \mathsf{e}'^{\div}\ \_\ \mathbf{id} : \mathsf{A}'^{+} \in \Gamma^{+}$, which follows by the same argument as Sub-Case Part 8 of the x case of Lemma 6.6.3.

Case: $(\lambda\,\mathsf{x}:\_.\,\mathsf{e}_1)\ \mathsf{e}_2 \rhd_{\beta} \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]$

We must show that $((\lambda\,\mathsf{x}:\_.\,\mathsf{e}_1)\ \mathsf{e}_2)^{\div} \rhd^{*} \mathbf{e}'$ and $\mathbf{e}' \equiv (\mathsf{e}_1[\mathsf{e}_2/\mathsf{x}])^{\div}$.

$$((\lambda\,\mathsf{x}:\_.\,\mathsf{e}_1)\ \mathsf{e}_2)$$

$$\begin{aligned}
&= (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}. && (131)\\
&\qquad (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,\mathbf{k}\ (\boldsymbol{\lambda}\,\mathbf{x}.\,e_1^{\div}))\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{f}.\,e_2^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x}.\,(\mathbf{f}\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k}))) &&\\
&\quad \text{by translation} &&\\
\rhd^* &\ (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,e_2^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x}.\,((\boldsymbol{\lambda}\,\mathbf{x}.\,e_1^{\div})\ \mathbf{x})\ \boldsymbol{\alpha}\ \mathbf{k})) && (132)\\
&\quad \text{by } \rhd_\beta &&\\
\rhd^* &\ (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,e_2^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x}.\,e_1^{\div}\ \boldsymbol{\alpha}\ \mathbf{k})) && (133)\\
&\quad \text{by } \rhd_\beta &&\\
&\equiv (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,(\boldsymbol{\lambda}\,\mathbf{x}.\,e_1^{\div}\ \boldsymbol{\alpha}\ \mathbf{k})\ (e_2^{\div}\ \_\ \mathbf{id})) && (134)\\
&\quad \text{by Rule } \equiv\text{-C}\textsc{ont} &&\\
\rhd^* &\ (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,(e_1^{\div}\ \boldsymbol{\alpha}\ \mathbf{k})[(e_2^{\div}\ \_\ \mathbf{id})/\mathbf{x}]) && (135)\\
&\quad \text{by } \rhd_\beta &&\\
&= (\boldsymbol{\lambda}\,\boldsymbol{\alpha}.\,\boldsymbol{\lambda}\,\mathbf{k}.\,(e_1^{\div}\ \boldsymbol{\alpha}\ \mathbf{k}))[(e_2^{\div}\ \_\ \mathbf{id})/\mathbf{x}] && (136)\\
&\quad \text{by substitution} &&\\
&\equiv e_1^{\div}[e_2^{\div}\ \_\ \mathbf{id}/\mathbf{x}] && (137)\\
&\quad \text{by Rule } \equiv\text{-}\eta &&\\
&= (e_1[e_2/x])^{\div} && (138)\\
&\quad \text{by Lemma 6.6.2} &&
\end{aligned}$$

□

Note that kinds do not take steps in the reduction relation, but can in the conversion relation.

Lemma 6.6.4 (CPS^v Preservation of Conversion)**.**

- *If* $\Gamma \vdash e : A$ *and* $e \rhd^* e'$ *then* $e^{\div} \rhd^* \mathbf{e'}$ *and* $\mathbf{e'} \equiv e'^{\div}$
- *If* $\Gamma \vdash A : K$ *and* $A \rhd^* A'$ *then* $A^{+} \rhd^* \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^{+}$
- *If* $\Gamma \vdash A : \star$ *and* $A \rhd^* A'$ *then* $A^{\div} \rhd^* \mathbf{A'}$ *and* $\mathbf{A'} \equiv A'^{\div}$
- *If* $\Gamma \vdash K : U$ *and* $K \rhd^* K'$ *then* $K^{+} \rhd^* \boldsymbol{\kappa'}$ *and* $\boldsymbol{\kappa'} \equiv K'^{+}$

Proof. The proof is straightforward by induction on the derivation of $t \rhd^* t'$.[4] □

Lemma 6.6.5 (CPS^v Preservation of Equivalence)**.**

- *If* $e \equiv e'$ *then* $e^{\div} \equiv e'^{\div}$
- *If* $A \equiv A'$ *then* $A^{\div} \equiv A'^{\div}$
- *If* $A \equiv A'$ *then* $A^{+} \equiv A'^{+}$
- *If* $K \equiv K'$ *then* $K^{+} \equiv K'^{+}$

4 In the previous version of this work (Bowman et al., 2018), this proof was incorrectly stated as by induction on the length of reduction sequences.

Lemma 6.6.6 (CPS^v Type and Well-formedness Preservation)**.**

1. *If* $\vdash \Gamma$ *then* $\vdash \Gamma^+$
2. *If* $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash e^\div : A^\div$
3. *If* $\Gamma \vdash A : K$ *then* $\Gamma^+ \vdash A^+ : K^+$
4. *If* $\Gamma \vdash A : \star$ *then* $\Gamma^+ \vdash A^\div : \star^+$
5. *If* $\Gamma \vdash K : U$ *then* $\Gamma^+ \vdash K^+ : U^+$

Proof. All cases are proven simultaneously by simultaneous induction on the type derivation and well-formedness derivation. Part 4 follows easily by part 3 in every case, so we elide its proof. Most cases follow easily from the induction hypotheses.

Case: Rule W-Assum $\vdash \Gamma, x : A$

There are two sub-cases: either A is a type or a kind.

Sub-case: A is a type

We must show $\vdash \Gamma^+, \mathbf{x} : A^+$.

It suffices to show that $\Gamma^+ \vdash A^+ : \boldsymbol{\kappa}$, which follows by part 3 of the induction hypothesis.

Sub-case: A is a kind; similar to the previous case, except the goal follows by part 5 of the induction hypothesis.

Case: Rule W-Def $\vdash \Gamma, x = e : A$

We give the case for when A is a type; the case when A is a kind is similar.

We must show $\vdash \Gamma^+, \mathbf{x} = e^\div\ A^+\ \mathbf{id} : A^+$.

It suffices to show that $\Gamma^+ \vdash e^\div\ A^+\ \mathbf{id} : A^+$.

By part 2 of the induction hypothesis and definition of the translation, we know that $\Gamma^+ \vdash e^\div : \boldsymbol{\Pi\,\alpha} : \boldsymbol{\star}. (A^+ \boldsymbol{\to \alpha}) \boldsymbol{\to \alpha}$, easily which implies the goal.

Case: Rule Var $\Gamma \vdash x : A$

We give the case for when A is a type; the case when A is a kind is simple since the translation on type variables is the identity.

We must show that $\Gamma^+ \vdash \boldsymbol{\lambda\,\alpha} : \boldsymbol{\star}. \boldsymbol{\lambda}\,\mathbf{k} : A^+ \boldsymbol{\to \alpha}. \mathbf{k}\ \mathbf{x} : A^\div$

By the part 1 of the induction hypothesis, we know $\Gamma^+ \vdash \mathbf{x} : A^+$, which implies the goal.

Case: Rule App $\Gamma \vdash e_1\ e_2 : B[e_2/x]$.

There are four sub-cases: e_1 can be either a term or a type, and e_2 can be either a term or a type. The interesting case is when both are terms, since this is the case most affected by the CPS translation.

Sub-case: CPS^v_e-App, both e_1 and e_2 are terms.

We must show that

$$\begin{array}{l}\lambda\, \alpha : \star.\, \lambda\, k : (B^+[(e_2^\div\ A^+\ \mathbf{id})/x]) \to \alpha. \\ \qquad e_1^\div\ \alpha\ (\lambda\, f : \Pi\, x : A^+.\, B^\div.\, e_2^\div\ @\ \alpha\ (\lambda\, x : A^+.\, (f\ x)\ \alpha\ k))\end{array}$$

has type $(B[e_2/x])^\div$.

Note that,

$$\begin{array}{llr} & (B[e_2/x]) & \\ \equiv & B^\div[e_2^\div\ A^+\ \mathbf{id}/x] & \text{by Lemma 6.6.2} \quad (139) \\ \equiv & \Pi\, \alpha : \star.\, ((B^+[e_2^\div\ A^+\ \mathbf{id}/x]) \to \alpha) \to \alpha & \text{by translation} \quad (140)\end{array}$$

Hence it suffices to show that

$$\Gamma^+, \alpha : \star, k : (B^+[(e_2^\div\ A^+\ \mathbf{id})/x]) \to \alpha \vdash e_1^\div\ \alpha\ (\lambda\, f : \Pi\, x : A^+.\, B^\div.\ e_2^\div\ @\ \alpha\ (\lambda\, x : A^+.\ (f\ x)\ \alpha\ k) : \alpha$$

By part 2 of the induction hypothesis, we know that

$$\Gamma^+ \vdash e_1^\div : \Pi\, \alpha : \star.\, ((\Pi\, x : A^+.\, B^\div) \to \alpha) \to \alpha,$$

hence it suffices to show that

$$\begin{array}{ll}\Gamma^+, \alpha : \star, & \vdash e_2^\div\ @\ \alpha\ (\lambda\, x : A^+.\, (f\ x)\ \alpha\ k) : \alpha \\ \quad k : (B^+[(e_2^\div\ A^+\ \mathbf{id})/x]) \to \alpha, & \\ \quad f : \Pi\, x : A^+.\, B^\div & \end{array}$$

By Rule T-Cont, we must show

$$\begin{array}{ll}\Gamma^+, \alpha : \star, & \vdash (f\ x)\ \alpha\ k : \alpha \\ \quad k : (B^+[(e_2^\div\ A^+\ \mathbf{id})/x]) \to \alpha, & \\ \quad f : \Pi\, x : A^+.\, B^\div, & \\ \quad x = e_2^\div\ A^+\ \mathbf{id}, & \end{array}$$

Note that $f\ x : B^\div[x/x]$ and $B^\div[x/x] = \Pi\, \alpha : \star.\, (B^+[x/x]) \to \alpha \to \alpha$.

But $k : (B^+[(e_2^\div\ A^+\ \mathbf{id})/x]) \to \alpha$.

Hence it suffices to show that $(B^+[x/x]) \equiv (B^+[(e_2^\div\ A^+\ \mathbf{id})/x])$, which follows by δ reduction on x since we have $x = e_2^\div\ A^+\ \mathbf{id}$ by Rule T-Cont.

Note that without the new typing rule, we would be here stuck. However, thanks to Rule T-Cont, we have the equality that $x = e_2^\div\ A^+\ \mathbf{id}$, and we are able to complete the proof.

Sub-case: e_1 is a term but e_2 is a type A'. This case is similar to the application case of the CBN translation. It does not require the new typing rule Rule T-Cont, as the argument is a type, the argument is not CPS translated.

Sub-case: e_1 is a type and e_2 is a term. This case is simple; note that the translate CPS^v_A-AppConstr translates the argument e_2 into $e_2^{\div}\ A^+\ \mathbf{id}$ since the term variable must have a value type.

Sub-case: Both e_1 and e_2 are types. This case is trivial by the induction hypothesis.

Case: Rule Let $\Gamma \vdash \mathsf{let}\, x = e_1 : A \,\mathsf{in}\, e_2 : B[e_1/x]$

There are four sub-cases, as in the case of application, and the proofs are nearly identical. This should be unsurprising, since the new typing rule Rule T-Cont essentially gives the typing of application of a continuation the same expressive power as dependent let. I give the case for both e_1 and e_2 are terms, since this is the most interesting case.

Sub-case: CPS^v_e-Let

We must show that $\Gamma^+ \vdash \boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\, \boldsymbol{\lambda}\,\mathbf{k} : B^+[e_2^{\div}\ A^+\ \mathbf{id}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}.\quad : (B[e_2/x])^{\div}$
$e_1^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x} : A^+.\, e_2^{\div}\ \boldsymbol{\alpha}\ \mathbf{k})$

By Lemma 6.6.2 and the definition of the translation, it suffices to show

$\Gamma^+, \boldsymbol{\alpha} : \star, \mathbf{k} : B^+[e_2^{\div}\ A^+\ \mathbf{id}/\mathbf{x}] \rightarrow \boldsymbol{\alpha} \vdash e_1^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{x} : A^+.\, e_2^{\div}\ \boldsymbol{\alpha}\ \mathbf{k}) : \boldsymbol{\alpha}$

By Rule T-Cont, we must show

$\Gamma^+, \boldsymbol{\alpha} : \star, \mathbf{k} : B^+[e_2^{\div}\ A^+\ \mathbf{id}/\mathbf{x}] \rightarrow \boldsymbol{\alpha}, \mathbf{x} = e_1^{\div}\ A^+\ \mathbf{id} \vdash e_2^{\div}\ \boldsymbol{\alpha}\ \mathbf{k} : \boldsymbol{\alpha}$

Note that by the induction hypothesis,

$\Gamma^+, \mathbf{x} = e_1^{\div}\ A^+\ \mathbf{id} \vdash e_2^{\div} : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\, (B^+ \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$

Hence by δ-reduction and Rule Conv,

$\Gamma^+, \mathbf{x} = e_1^{\div}\ A^+\ \mathbf{id} \vdash e_2^{\div} : \boldsymbol{\Pi}\,\boldsymbol{\alpha} : \star.\, (B^+[(e^{\div}\ A^+\ \mathbf{id})/\mathbf{x}] \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$ which implies the goal.

Case: Rule Pair

Note that the translation of dependent pairs, CPS^v_e-Pair, also requires a use of the rule Rule T-Cont. Since the source language allows pairs of expressions, but our target language for the CBV translation should not, we must evaluate both components of the pair before calling the continuation. However, since the type of second component depends on the value of the first component, we must apply Rule T-Cont when typing the application of the continuation to the first component so that we have $\mathbf{x}_1 = e_1^{\div}\ A^+\ \mathbf{id}$ when typing the continuation for the second component.

The proof is similar to the case for Rule App.

Case: Rule Snd The proof is exactly like the case for the CBN translation. □

Theorem 6.6.7 (CPS^v Type Preservation). *If* $\Gamma \vdash e : A$ *then* $\Gamma^+ \vdash \text{cps}\,[\![e]\!] : \text{cpsT}\,[\![A]\!]$.

6.6.2 Compiler Correctness

To prove correctness of separate compilation for CPS^v, I follow the standard architecture from Chapter 3 and used in Section 6.5.2. I use the same cross-language relation $\approx$ on observation. However, note that in CBV, we should only link with values, so I restrict closing substitutions γ to values and use the value translation on substitutions γ^+. The proofs follow exactly the same structure as in Section 6.5.2.

Theorem 6.6.8 (Separate Compilation Correctness). *If* $\Gamma \vdash \mathsf{e}$ *and* $\Gamma \vdash \gamma$*, then* $\mathsf{eval}(\gamma(\mathsf{e})) \approx \mathbf{eval}(\gamma^+(\mathsf{e}^{\div}))$.

Corollary 6.6.9 (Whole-Program Correctness). *If* $\vdash \mathsf{e}$ *then* $\mathsf{eval}(\mathsf{e}) \approx \mathbf{eval}(\mathsf{e}^{\div})$.

6.7 Related and Future Work

DEPENDENT CONDITIONALS In addition to showing that the traditional double-negation CPS for Σ types is not type preserving, Barthe and Uustalu (2002) demonstrate an impossibility result for CPS and dependent conditionals. They prove that no type-correct CPS translation can exist for sums with dependent case. However, careful inspection of their proof reveals that this impossibility result only applies if the CPS translation allows unrestricted control effects. As my translation does not allow control effects, I conjecture it is possible to prove type preservation for dependent conditionals.

The impossibility result by Barthe and Uustalu relies on the ability to implement `call/cc` via the CPS translation. Assuming there is a type-preserving CPS translation, they construct a model of CoC extended with `call/cc` and sum types (CCΔ+) in CoC with sum types (CoC+). Since CoC+ is consistent, this model proves that CoCΔ+ is consistent. However, it is known that CoCΔ+ is inconsistent (Coquand, 1989). Therefore, the type-preserving CPS translation of CoCΔ+ cannot exist.

Their proof is valid; however, it is well known that the polymorphic answer type CPS translation that I use cannot implement `call/cc` (Ahmed and Blume, 2011). Therefore, my translation does not give a model of CoCΔ+ in CoC+.

I further conjecture that my CPS translation is type preserving for dependent conditionals. Since the original work by Barthe and Uustalu uses sums with dependent case, I use these instead of the dependent if presented in Chapter 2, but the same core idea applies to either implementation of dependent conditionals. The typing rule for dependent case is the following.

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A} + \mathsf{B} \qquad \begin{array}{c}\Gamma, \mathsf{y} : \mathsf{A} + \mathsf{B} \vdash \mathsf{A}' : \star \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e}_1 : \mathsf{A}'[\mathsf{inj}_1\, \mathsf{x}/\mathsf{y}]\end{array} \qquad \Gamma, \mathsf{x} : \mathsf{B} \vdash \mathsf{e}_2 : \mathsf{A}'[\mathsf{inj}_2\, \mathsf{x}/\mathsf{y}]}{\Gamma \vdash \mathsf{case}\, \mathsf{e}\, \mathsf{of}\, \mathsf{x}.\, \mathsf{e}_1 ; \mathsf{x}.\, \mathsf{e}_2 : \mathsf{A}'[\mathsf{e}/\mathsf{y}]}$$

I would extend the CPS^n with the following rule.

$$\begin{aligned}(\mathsf{case}\,\mathsf{e}\,\mathsf{of}\,\mathsf{x}.\,\mathsf{e}_1;\mathsf{x}.\,\mathsf{e}_2)^{\div} = {} & \boldsymbol{\lambda}\,\boldsymbol{\alpha} : \star.\,\boldsymbol{\lambda}\,\mathbf{k} : \mathsf{A}'^{+}[\mathsf{e}^{\div}/\mathbf{y}] \rightarrow \boldsymbol{\alpha}.\\ & \mathsf{e}^{\div}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\,\mathbf{y} : \mathsf{A}^{+} + \mathsf{B}^{+}.\,\mathbf{case}\,\mathbf{y}\,\mathbf{of}\,\mathbf{x}.\,(\mathsf{e}_1^{\div}\ \boldsymbol{\alpha}\ \mathbf{k});\mathbf{x}.\,(\mathsf{e}_2^{\div}\ \boldsymbol{\alpha}\ \mathbf{k}))\end{aligned}$$

Focusing on the first branch of the **case** above, note that $\mathsf{e}_1^{\div}\ \boldsymbol{\alpha} : (\mathsf{A}'^{+}[(\mathsf{inj}_1\,\mathsf{x})^{\div}/\mathbf{y}] \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$, however $\mathbf{k} : \mathsf{A}'^{+}[\mathsf{e}^{\div}/\mathbf{y}] \rightarrow \boldsymbol{\alpha}$. We need to show that $\mathsf{e}^{\div} \equiv (\mathsf{inj}_1\,\mathsf{x})^{\div}$, similar to the problem with the second projection of dependent pairs. This time, however, to show $\mathsf{e}^{\div} \equiv (\mathsf{inj}_1\,\mathsf{x})^{\div}$ we need to reason about the flow of the underlying value of $\mathsf{e}^{\div}$ into $\mathbf{y}$ and also about the relationship between $\mathbf{y}$ and $\mathbf{x}$. Specifically, we need to first use my new T-CONT rule, which allows us to assume $\mathbf{y} = \mathsf{e}^{\div}\ \boldsymbol{\alpha}\ \mathbf{id}$. Next, we need to know that since the case analysis is on the value $\mathbf{y}$, in the first branch $\mathbf{y} \equiv \mathbf{inj}_1\,\mathbf{x}$ (and similarly for the other branch), but the problem is that the existing typing rule for dependent case does not let us assume that.

Nonetheless, I conjecture the same extension I propose in Chapter 4 works to allow type preservation for CPS. We simply change the typing rule for case to record the equality $\mathsf{e} \equiv \mathsf{inj}_1\,\mathsf{x}$ while typing the first branch, and similarly for the second branch, like in the following rule.

$$\frac{\begin{array}{c}\Gamma, \mathsf{y} : \mathsf{A} + \mathsf{B} \vdash \mathsf{A}' : \star \qquad \Gamma \vdash \mathsf{e} : \mathsf{A} + \mathsf{B}\\ \Gamma, \mathsf{x} : \mathsf{A}, \mathsf{p} : \mathsf{e} = \mathsf{inj}_1 \vdash \mathsf{e}_1 : \mathsf{A}'[\mathsf{inj}_1\,\mathsf{x}/\mathsf{y}] \qquad \Gamma, \mathsf{x} : \mathsf{B}, \mathsf{p} : \mathsf{e} = \mathsf{inj}_2 \vdash \mathsf{e}_2 : \mathsf{A}'[\mathsf{inj}_2\,\mathsf{x}/\mathsf{y}]\end{array}}{\Gamma \vdash \mathsf{case}\,\mathsf{e}\,\mathsf{of}\,\mathsf{x}.\,\mathsf{e}_1;\mathsf{x}.\,\mathsf{e}_2 : \mathsf{A}'[\mathsf{e}/\mathsf{y}]}$$

With this modification in the target language CoC^k, we can type check the above translation of dependent case. I have not yet shown the consistency of the target language CoC^k once it is extended with this modified typing rule for dependent case, primarily due to problems scaling the CPS translation to higher universes, which I discuss next, and further in Chapter 8.

HIGHER UNIVERSES It is unclear how to scale this CPS translation to higher universes. In this work, I have a single impredicative universe $\star$, and the locally polymorphic answer type $\boldsymbol{\alpha}$ lives in that universe. With an infinite hierarchy, it is not clear what the universe of $\boldsymbol{\alpha}$ should be. Furthermore, the translation relies on impredicativity in $\star$. We can only use impredicativity at one universe level and the locally polymorphic answer-type translation does not seem possible in the context of predicative polymorphism, so it's unclear how to adapt our translation to the infinite predicative hierarchy.

I initially conjectured that the right solution for handling universes was universe polymorphism (Sozeau and Tabareau, 2014). My thought was that since the type is provided by the calling context, it seems sensible that the calling context should also provide the universe. Then, we could modify the type translation to be $\boldsymbol{\Pi}\,\ell : \mathbf{Level}.\,\boldsymbol{\Pi}\,\boldsymbol{\alpha} : \mathbf{Type}_{\ell}.\,(\mathsf{A}^{+} \rightarrow \boldsymbol{\alpha}) \rightarrow \boldsymbol{\alpha}$. However, one of the authors of Sozeau and Tabareau (Sozeau) suggested to me that this would not work.[5]

5 Personal communication. Jan. 2018.

CONTROL OPERATORS AND DEPENDENT TYPES This chapter explicitly avoids control effects and dependent types to focus on type preservation. However, in general, we may want to combine the two. Herbelin (2005) shows that unrestricted use of `call/cc` and `throw` in a language with Σ types and equality leads to an inconsistent system. The inconsistency is caused by type dependency on terms involving control effects. Herbelin (2012) solves the inconsistency by constraining types to depend only on *negative-elimination-free (NEF)* terms, which are free of effects. This restriction makes dependent types compatible with classical reasoning enabled by the control operators.

After the initial publication of the work in this chapter (Bowman et al., 2018), Cong and Asai (2018a) extended the CPS translation to control effects in the NEF fragment, and use this locally polymorphic translation to disallow control effects for terms that use unsafe dependencies.

This is similar to recent work by Miquey (2017) uses the NEF restriction to soundly extend the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin (2000), a computational classic calculus, with dependent types. He then extends the language with delimited continuations, and defines a type-preserving CPS to a standard intuitionistic dependent type theory. By comparison, our source language CoC^D is effect-free, therefore we do not need the NEF condition to restrict dependency. My use of the identity function serves a similar role to their delimited continuations—allowing local evaluation of a CPS'd computation.

EFFECTS AND DEPENDENT TYPES Pédrot and Tabareau (2017) define the *weaning translation*, a monadic translation for adding effects to dependent type theory. The weaning translation allows us to represent self-algebraic monads, *i.e.*, monads whose algebras' universe itself forms an algebra, such as exception and non-determinism monads. However, it does not apply to the standard continuation monad, which is not self-algebraic. The paper extends the translation to inductive types, with a restricted form of dependent elimination. Full dependent elimination can be implemented only for terms whose type is first-order, and this comes at the cost of inconsistency, although one can recover consistency by requiring that every inductive term be parametric. My translation does not lead to inconsistency, and requires no restrictions on the type to be translated. However, my translation appears to impose the self-algebraic structure on the computation types, and my use of parametricity to cast computations to values is similar to their parametricity restriction.

INTERNALIZING PARAMETRICITY My work internalizes a specific free theorem, but ongoing work focuses on how to internalize parametricity more generally in a dependent type theory. Krishnaswami and Dreyer (2013) develop a technique for adding new rules to the extensional CoC. They present a logical relation for terms that are not syntactically well typed, but are semantically well behaved and equivalent at a particular type. Using this logical relation, they prove the consistency of several extensions to extensional CoC, including sum types, dependent records, and natural numbers. Bernardy et al. (2012); Keller and Lasson (2012) give translations from one dependent type theory into another that yield a parametric model of the original theory. These essentially encode the logical relation in the target type theory, similar to the approach we took in Section 6.4.1 (Meta-theory). Recent work by Nuyts et al. (2017) develops a theory that internalizes parametricity, including the important concept of *identity extension*, and gives a thorough comparison of the literature. By building a target language based on one of these systems, it's possible that we could eliminate the rule ≡-CONT as an axiom and instead derive it internally. This could allow the translation to scale to theories that are orthogonal to parametricity.

PERVASIVE TRANSLATION In this chapter, I only CPS translate *terms*, *i.e.*, run-time expressions. However, other work (Barthe et al., 2001) studies *pervasive* translation of PTSs. A pervasive CPS translation internalizes evaluation order for all expressions: terms, types, and kinds. This is necessary in general, since in a language such as Coq, we cannot easily distinguish between terms, types, and kinds. A pervasive translation may be necessary to scale type-preserving translation to higher universes and more type-level computation. The translations in Chapter 4 and Chapter 5 both use pervasive translation for this reason.

A pervasive CPS translation is also used in a partial solution to the Barendregt-Geuvers-Klop conjecture (Geuvers, 1993) which essentially states that every weakly normalizing PTS is also strongly normalizing. The conjecture has been solved for non-dependent PTSs, but the solution for dependent PTSs remains open.

7 PARAMETRIC CLOSURE CONVERSION

In this chapter, I develop a *parametric closure conversion* translation, *i.e.*, a closure conversion translation that does not add primitive closures to the target language but instead encodes them using existential types. As discussed in Chapter 5, this translation is not suitable in all dependent type theories, since it relies on parametricity and impredicativity. However, parametric closure conversion has some advantages over abstract closure conversion which makes studying this translation worthwhile. In particular, known optimizations for recursive closures that avoid rebuilding the closure every time through the loop use parametric closure conversion (Minamide et al., 1996; Morrisett and Harper, 1998). We can develop a parametric closure conversion if we admit impredicativity and parametricity, and avoid higher universes.

I begin with a review of the problems with parametric closure conversion discussed in Chapter 5. To accommodate the restrictions, the parametric closure conversion translation is defined on CoC^D, the same source language used in Chapter 6, and a restriction of ECC^D presented in Chapter 2. I then design a target language, CoC^{CC}, before developing the full translation, and proofs of type preservation and compiler correctness.

Typographical Note. *In this chapter, I typeset the source language, CoC^D, in a* blue, non-bold, sans-serif font, *and the target language, CoC^{CC}, in a* **bold, red, serif font**.

7.1 Main Ideas

As review, recall from Chapter 5 the problem with preserving dependent types through closure conversion using the existential types. I reproduce the example from Chapter 5 below.

$$\lambda\, \mathsf{A} : \star.\, \lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{x} \quad : \quad \Pi\, \mathsf{A} : \star.\, \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{A}$$

Here we have the polymorphic identity function, where the first argument A, a type, is free in the second function and is used in the type of the function. The standard parametric closure conversion translation of this example is below, reproduced from Chapter 5.

$$\begin{aligned}&\langle\!\langle \boldsymbol{\lambda}\,(\mathbf{n_1} : \mathbf{1}, \mathbf{A} : \mathbf{Prop}).\, \langle\!\langle \boldsymbol{\lambda}\,(\mathbf{n_2} : \mathbf{Prop} \times \mathbf{1}, \mathbf{x} : \mathbf{fst\,n_2}).\, \mathbf{x}, \langle \mathbf{A}, \langle\rangle\rangle\rangle\!\rangle, \langle\rangle\rangle\!\rangle \quad : \\ &\quad \exists\, \boldsymbol{\alpha}_1 : \mathbf{Prop}\,.\,(\boldsymbol{\Pi}\,(\mathbf{n_1} : \boldsymbol{\alpha}_1, \mathbf{A} : \mathbf{Prop}). \\ &\qquad\qquad\qquad \exists\, \boldsymbol{\alpha}_2 : \mathbf{Type}\,.\,(\boldsymbol{\Pi}\,(\mathbf{n_2} : \boldsymbol{\alpha}_2, \mathbf{x} : \mathbf{fst\,n_2}).\, \mathbf{fst\,n_2}) \times \boldsymbol{\alpha}_2) \times \boldsymbol{\alpha}_1\end{aligned}$$

Unfortunately, while the code is well-typed, the closure is not. Recall that the code of the inner closure has type $\boldsymbol{\Pi}\,(\mathbf{n}_2 : \boldsymbol{\alpha}_2, \mathbf{x} : \mathbf{fst}\,\mathbf{n}_2).\,\mathbf{fst}\,\mathbf{n}_2$. In the type of the closure, this projection $\mathbf{fst}\,\mathbf{n}_2$ is not well-typed since the type of the environment is hidden—$\mathbf{fst}\,\mathbf{n}_2$ is invalid when $\mathbf{n}_2$ has type $\boldsymbol{\alpha}_2$.

A similar problem presents itself in the type-preserving closure conversion of System F by Minamide et al. (1996). In that work, type variables in System F are included in the environment, introducing the same problem as above of closures not being well-typed. In subsequent work, Morrisett et al. (1999) observed that in System F type variables can be considered computationally irrelevant (and will, therefore, be erased before runtime). This justified a simple translation where closure-converted functions were allowed to have free type (though not term) variables. In dependently typed languages, this solution does not apply since term variables can also appear in types.

As before in Chapter 5, and in the work of Minamide et al. (1996), we have a dilemma. We must close the type of code since code must be closed after closure conversion, but we must also leave the type of the closure open since we cannot project from the hidden environment. The challenge is to unify the closed type of the code and the open type of the closure.

In this chapter, I adapt the parametric closure conversion of Minamide et al. (1996) to dependent types. Previously, in Chapter 5, I essentially axiomatize the target language with closures, adapting the abstract closure conversion of Minamide et al. (1996) to dependent types. The primitive closure captures exactly the above needs. I use existential types to encode closures, and use a form of singleton types, called translucent types, to encode an equality between the hidden environment and the free variables. The translucent type unifies the hidden environment with the type variables, thus unifying the closed type of the closure and the open type of the code.

Below is the nearly complete translation to demonstrate how this unifies the two seemingly different types.

Notice that this type translation relies on impredicativity, since the existential type must be in the same universe as the type variable α over which it quantifies.

$$
\begin{aligned}
[\![(\Pi\, x : A.\, B)]\!] &= \exists\, \boldsymbol{\alpha} : \star.\, (\boldsymbol{\alpha} \times \mathbf{Code}\, \mathbf{n} : \boldsymbol{\alpha}, \mathbf{x} : A^{+}.\, [\![B]\!]) \\
[\![(\lambda\, x : A.\, e)]\!] &= \mathbf{pack}\,\langle\langle\langle \mathbf{x}_i \ldots\rangle, \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\,\mathbf{in}\,[\![A]\!]).\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\,\mathbf{in}\,[\![e]\!]\rangle, \mathbf{A}'\rangle \\
&\quad \text{where } \mathbf{A}' = \boldsymbol{\Sigma}\, \mathbf{x}_i : [\![A_i]\!] \ldots \\
&\qquad\quad x_i : A_i \ldots \text{ are the free variables of } e \text{ and } A
\end{aligned}
$$

This translation uses the code type as in Chapter 5 to distinguish closed code from closures, but lacks a Π-type since closures are encoded using existential types. Using this translation, we need to complete the following derivation in order to prove type preservation. As we will see, we cannot complete the derivation for the standard existential type translation.

$$
\dfrac{\dfrac{\text{stuck}}{\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A} \vdash \mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\,\mathbf{in}\,\mathbf{e} : [\![B]\!]}}{\boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{let}\,\langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\,\mathbf{in}\,\mathbf{e} : \mathbf{Code}\,\mathbf{n} : \mathbf{A}', \mathbf{x} : [\![A]\!].\,[\![B]\!]}
$$

I focus on showing the code is well-typed at the translated type, since that is where the problem arises. Notice that $[\![B]\!]$ has free variables in the type. To complete this derivation, we must show $\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e} : [\![B]\!]$. However, by the typing rule for dependent let, we can only conclude $\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e} : [\![B]\!][\boldsymbol{\pi}_i\ \mathbf{n}/\mathbf{x}_i]$. We cannot complete the derivation and this translation is not type preserving because dependent types allow types to depend on free term variables from the environment.

However, intuitively, it should be the case that $[\![B]\!] \equiv [\![B]\!][\boldsymbol{\pi}_i\ \mathbf{n}/\mathbf{x}_i]$. By parametricity, we know $\mathbf{n}$ will only get replaced by exactly the environment generated by our compiler. Since the compiler generates the environment $\mathbf{n} = \langle \mathbf{x}_i \ldots \rangle$, this substitution is essentially a no-op, replacing $\mathbf{x}_i$ by $\mathbf{x}_i$. To develop a type-preserving translation, we need to find a way to internalize this reasoning so that the type system can decide this equivalence.

This is why we rely on parametricity, and where the type preservation argument fails if we cannot have parametricity in the target language.

To solve this, I use a form of singleton types called translucent types (Harper and Lillibridge, 1994; Lillibridge, 1997) to encode this equivalence in a type. The *translucent function type*[1], written $\mathbf{e}' \Rightarrow \mathbf{B}$, represents a function (or in our case, code) that must be applied to (an expression equivalent to) the term $\mathbf{e}'$ and produces something of type $\mathbf{B}$. The translucent function type rules are the following. Essentially, any function $\mathbf{f}$ can be statically specialized to a particular argument $\mathbf{e}'$. This has the effect of instantiating the dependent function type before the function is actually applied. After that, it must be dynamically applied to that argument.

$$\frac{\boldsymbol{\Gamma} \vdash \mathbf{f} : \boldsymbol{\Pi}\,\mathbf{n} : \mathbf{A}'.\,\mathbf{B} \qquad \boldsymbol{\Gamma} \vdash \mathbf{e}' : \mathbf{A}'}{\boldsymbol{\Gamma} \vdash \mathbf{f} : \mathbf{e}' \Rightarrow \mathbf{B}[\mathbf{e}'/\mathbf{n}]}\ \textsc{TrFun} \qquad \frac{\boldsymbol{\Gamma} \vdash \mathbf{f} : \mathbf{e}' \Rightarrow \mathbf{B}}{\boldsymbol{\Gamma} \vdash \mathbf{f}\ \mathbf{e}' : \mathbf{B}}\ \textsc{TrApp}$$

To encode closures, we existentially quantify over the *value* of the environment $\mathbf{n}$, in addition to the type of the environment $\boldsymbol{\alpha}$, leveraging dependent types. Then we describe type of the code $\mathbf{f}$ using a translucent type $(\mathbf{n} \Rightarrow \mathbf{Code}\,\mathbf{x} : [\![A]\!].\,[\![sB]\!])$, which requires that the code be applied to exactly the environment $\mathbf{n}$ and an arbitrary argument $\mathbf{x}$ of type $[\![A]\!]$, and produces an output of type $[\![B]\!]$. The translation becomes the following.

$$\begin{aligned}
[\![(\Pi\, x : A.\, B)]\!] &= \exists\,(\boldsymbol{\alpha} : \star, \mathbf{n} : \boldsymbol{\alpha}, \mathbf{f} : (\mathbf{n} \Rightarrow \mathbf{Code}\,\mathbf{x} : [\![A]\!].\,[\![B]\!])). \\
[\![(\lambda\, x : A.\, e)]\!] &= \mathbf{pack}\,\langle \mathbf{A}', \langle \mathbf{x}_i \ldots \rangle, \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![A]\!]). \\
&\qquad\qquad \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ [\![e]\!]\rangle \\
&\text{where } \mathbf{A}' = \boldsymbol{\Sigma}\,(\mathbf{x}_i : [\![A_i]\!] \ldots) \\
&\qquad x_i : A_i \ldots \ \text{are the free variables of } e \text{ and } A
\end{aligned}$$

Now, the environment is still hidden from the client of the closure, but when checking that the closure is well-typed, the variable $\mathbf{n}$ is replaced by the value of the closure. Then all we need is to prove $[\![B]\!] \equiv [\![B]\!][\boldsymbol{\pi}_i\ \langle \mathbf{x}_i \ldots \rangle/\mathbf{x}_i]$, which is trivial. This was essentially the insight in Minamide et al. (1996), although in a simpler setting.

1 This particular presentation of translucent function types is due to Minamide et al. (1996).

$$
\begin{array}{llcl}
\textit{Universes} & \mathbf{U} & ::= & \star \mid \square \\
\textit{Expressions} & \mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \star \mid \mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{B} \\
 & & \mid & \mathbf{e} \Rightarrow \mathbf{Code}\,\mathbf{n} : \mathbf{A}'.\,\mathbf{B} \mid \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{e} \mid \mathbf{e}_f\ \mathbf{e}_1\ \mathbf{e}_2 \\
 & & \mid & \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \mid \langle \mathbf{e}, \mathbf{e} \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \mid \mathbf{fst}\,\mathbf{e} \mid \mathbf{snd}\,\mathbf{e} \\
 & & \mid & \mathbf{let}\,\mathbf{x} = \mathbf{e} : \mathbf{A}\ \mathbf{in}\,\mathbf{e} \mid \exists\,(\alpha : \mathbf{A}, \mathbf{n} : \mathbf{A}').\,\mathbf{B} \\
 & & \mid & \mathbf{pack}\,\langle \mathbf{A}, \mathbf{e}', \mathbf{e} \rangle\ \mathbf{as}\,\exists\,(\alpha : \mathbf{A}, \mathbf{n} : \mathbf{A}').\,\mathbf{B} \\
 & & \mid & \mathbf{unpack}\,\langle \mathbf{x}, \mathbf{x}, \mathbf{x} \rangle = \mathbf{e}\ \mathbf{in}\,\mathbf{e} \mid \cdots \\
\textit{Environments} & \boldsymbol{\Gamma} & ::= & \cdot \mid \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A} \mid \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}
\end{array}
$$

Figure 7.1: CoC^{CC} Syntax (excerpts)

As we descend into the **pack** expression, the existentially quantified environment flows into the type and the translucent type yields an additional equivalence. The completed derivation is the following.

$$
\dfrac{\vdots \qquad \dfrac{\dfrac{\vdots}{\boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : [\![\mathsf{A}]\!]).\,\mathbf{e}' : \mathbf{Code}\,\mathbf{n} : \mathbf{A}', \mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!][\boldsymbol{\pi}_i\,\mathbf{n}/\mathbf{x}_i]} \qquad \dfrac{\text{trivial, since } \mathit{env} = \langle \mathbf{x}_i \ldots \rangle}{[\![\mathsf{B}]\!][\boldsymbol{\pi}_i\,\mathbf{n}/\mathbf{x}_i][\mathit{env}/\mathbf{n}] \equiv [\![\mathsf{B}]\!]}}{\boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : [\![\mathsf{A}]\!]).\,\mathbf{let}\,\langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\,\mathbf{in}\,[\![\mathsf{e}]\!] : \mathit{env} \Rightarrow \mathbf{Code}\,\mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!]}}{\boldsymbol{\Gamma} \vdash \mathbf{pack}\,\langle \mathbf{A}', \mathit{env}, \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : [\![\mathsf{A}]\!]).\,\mathbf{e}' \rangle : \exists\,\alpha : \star, \mathbf{n} : \alpha, \mathbf{f} : (\mathbf{n} \Rightarrow \mathbf{Code}\,\mathbf{x} : [\![\mathsf{A}]\!].\,).\,[\![\mathsf{B}]\!]}
$$

While the translation is similar in essence to Minamide et al. (1996), dependent types introduce additional difficulties. The above translation disrupts the syntactic reasoning about functions that the type system relies on, in particular, η-equivalence of functions.

To preserve η-equivalence, as in Chapter 5, I develop a principle of equivalence for closures. Essentially, closures are equivalent when we compute equivalent results under equivalent environments. This gives us an η-principle for closures that extends the η-principle for functions to include the environment. This principle also relies on parametricity to justify.

7.2 Parametric Closure Conversion IL

In Figure 7.1, I present the syntax of CoC^{CC}, a dependently typed closure converted language based on CoC^{D}. As in Chapter 5, closed code is written as $\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{e}$ and has type $\mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{B}$. All code takes two arguments, which are intuitively the environment $\mathbf{n}$ and the original argument $\mathbf{x}$. I add 2-ary existential types $\exists\,(\alpha : \mathbf{A}, \mathbf{n} : \mathbf{A}').\,\mathbf{B}$ to encode closures, with the introduction form $\mathbf{pack}\,\langle \mathbf{A}, \mathbf{e}', \mathbf{e} \rangle\ \mathbf{as}\,\exists\,(\alpha : \mathbf{A}, \mathbf{n} : \mathbf{A}').\,\mathbf{B}$ and the elimination form $\mathbf{unpack}\,\langle \alpha, \mathbf{n}, \mathbf{f} \rangle = \mathbf{e}'\ \mathbf{in}\,\mathbf{e}$. As with dependent pairs, I omit the type annotation on **pack** for brevity, as in $\mathbf{pack}\,\langle \mathbf{A}, \mathbf{e}', \mathbf{e} \rangle$. Finally, I add a variant of *translucent function*

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

$$\cdots \qquad \frac{\Gamma \vdash B : \star}{\Gamma \vdash \mathbf{Code}\,(x' : A', x : A).\, B : \star}\ \textsc{Code-*}$$

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

$$\frac{\Gamma \vdash e : \mathbf{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e : e' \Rightarrow \mathbf{Code}\,(x : A[e'/x']).\, B[e'/x']}\ \textsc{TrFun}$$

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

$$\frac{\Gamma \vdash f : e' \Rightarrow \mathbf{Code}\, x : A.\, B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e'\ e : B[e/x]}\ \textsc{TrApp}$$

$$\frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash A' : U' \qquad \Gamma, x : A, x' : A' \vdash B : \star}{\Gamma \vdash \exists\,(x : A, x' : A').\, B : \star}\ \textsc{Exist}$$

$$\frac{\begin{array}{c}\Gamma \vdash \exists\,(x : A, x' : A').\, B : U \\ \Gamma \vdash e : A \qquad \Gamma \vdash e' : A'[e/x] \qquad \Gamma \vdash e_b : B[e/x][e'/x']\end{array}}{\Gamma \vdash \mathbf{pack}\,\langle e, e', e_b \rangle\ \mathbf{as}\, \exists\,(x : A, x' : A').\, B : \exists\,(x : A, x' : A').\, B}\ \textsc{Pack}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \exists\,(x : A, x' : A').\, B \\ \Gamma, x : A, x' : A', x_b : B \vdash e' : B' \qquad \Gamma \vdash B' : U\end{array}}{\Gamma \vdash \mathbf{unpack}\,\langle x, x', x_b \rangle = e\ \mathbf{in}\, e' : B'}\ \textsc{Unpack}$$

Figure 7.2: CoC^{CC} Typing (excerpts)

types, specialized to code types, written $e \Rightarrow \mathbf{Code}\, x : A.\, B$. Since all code is 2-ary, the translucent function type is specialized to only curry code types. This type represents a code that can only be applied to the term e as its first argument, and something of type A as its second argument, and results in a term of the type B. Note that this type is not dependent in the usual sense on its first argument—*i.e.*, there is no name bound for the term e—since e has already been substituted into the types A and B.

We could simplify the type system by encoding 2-ary constructs such as ***code*** *and* ***pack*** *the same way we encode n-ary dependent pairs. However, this complicates η-equivalence for closures, which relies on the canonical forms.*

In Figure 7.2, I define the key typing rules for the target language; the complete definition is given in Appendix J. I explain translucent function types in detail shortly. I add the standard rules for existential types. Notice that existential types are impredicative (Rule Exist); unlike dependent pairs (strong dependent pairs with projection), existential types (weak dependent pairs with pattern matching) are consistent with impredicativity. Furthermore, impredicativity is necessary for expressing the type of

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

$$\vdots$$

$$\mathbf{unpack}\,\langle \mathbf{x}, \mathbf{x}', \mathbf{x}_b \rangle = \mathbf{pack}\,\langle \mathbf{e}, \mathbf{e}', \mathbf{e}_b \rangle\,\mathbf{in}\,\mathbf{e}_2 \quad \rhd_\exists \quad \mathbf{e}_2[\langle \mathbf{e}, \mathbf{e}', \mathbf{e}_b \rangle / \langle \mathbf{x}, \mathbf{x}', \mathbf{x}_b \rangle]$$

Figure 7.3: CoC^{CC} Reduction (excerpts)

closures. Rule TrFun is the introduction rule for the translucent type. Intuitively, any function can be statically specialized to a specific argument. To ascribe a term a translucent type, we must check that the specific argument is of the same type as expected by the code. Rule TrApp eliminates a translucent type. When applying code ascribed a translucent type, we check that the first argument is *equal* to the argument specified by the translucent type.

As an example of the translucent type, consider the following example in which we ascribe two different types to the polymorphic identity function.

$$\mathbf{id} : \mathbf{Code}\,(\mathbf{A} : \star, \mathbf{x} : \mathbf{A}).\,\mathbf{A}$$
$$\mathbf{id} : \mathbf{bool} \Rightarrow \mathbf{Code}\,\mathbf{x} : \mathbf{bool}.\,\mathbf{bool}$$

In the first case, we ascribe $\mathbf{id}$ the type $\mathbf{Code}\,(\mathbf{A} : \star, \mathbf{a} : \mathbf{A}).\,\mathbf{A}$. This is the standard type, compiled to a code type. In the second case, we ascribe $\mathbf{id}$ a translucent type. Notice that we do not actually apply $\mathbf{id}$ to any arguments; we specialize only its type. By ascribing it a translucent type, we know statically that when this function is applied to the type $\mathbf{bool}$, it accepts and return a $\mathbf{bool}$. This gives us the ability to reason about the result of applying $\mathbf{id}$ before applying it. As a result, after ascribing this type we *must* only apply $\mathbf{id}$ to $\mathbf{bool}$, or the additional static reasoning would not be valid.

In Figure 7.3 I give the new reduction rule for existential types. The rule is completely standard. Translucent types do not introduce new dynamic semantics.

Digression. *If we had recursion, and therefore recursive closures, we would need a non-standard reduction rule for* $\mathbf{unpack}$ *to ensure normalization. In the same way that recursive functions in Coq only reduce when applied to a constant, a recursive closure should only reduce when* $\mathbf{unpack}$*ed and applied to a constant.*

In Figure 7.4 I present new η-equivalence rules for closures. These new rules are the result of closure converting the η-equivalence rules from CoC^D.

This η-equivalence is specialized to closures and necessarily differs from the standard η-equivalence for existential types. To see why, consider the normal η-equivalence for existential types.

$$C[\mathbf{e}] \equiv \mathbf{unpack}\,\langle \alpha, \mathbf{x} \rangle = \mathbf{e}\,\mathbf{in}\,C[\mathbf{pack}\,\langle \mathbf{x}, \alpha \rangle]$$

This states that the expression $\mathbf{e}$ in some arbitrary program *context* (a program with a *hole*, or single linear variable, $[\cdot]$) C is equivalent to $\mathbf{unpack}$ing $\mathbf{e}$ and executing C

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

$$\ldots$$

$$\frac{\Gamma \vdash e_2 \rhd^* e_2' \qquad \begin{array}{c}\Gamma \vdash e_1 \rhd^* \mathbf{pack}\,\langle A', e', \lambda\,(n : A', x : A).\, e_1' \rangle \\ \Gamma, x : A[e'/n] \vdash e_1'[e'/n] \equiv \mathbf{unpack}\,\langle \alpha, n, f \rangle = e_2'\,\mathbf{in}\, f\ n\ x\end{array}}{\Gamma \vdash e_1 \equiv e_2} \equiv\text{-}\eta_1$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* \mathbf{pack}\,\langle A', e', \lambda\,(n : A', x : A).\, e_2' \rangle \\ \Gamma, x : A[e'/n] \vdash \mathbf{unpack}\,\langle \alpha, n, f \rangle = e_1'\,\mathbf{in}\, f\ n\ x \equiv e_2'[e'/n]\end{array}}{\Gamma \vdash e_1 \equiv e_2} \equiv\text{-}\eta_2$$

Figure 7.4: CoC^{CC} Equivalence (excerpts)

with the canonical form of the existential type $\mathbf{pack}\,\langle x, \alpha \rangle$. In a closure-converted language, we can't use this rule, since C may be code $\lambda\, y : A.\, [\cdot]$, and we can't push this context under the scope of the free variables α, x. In general, after closure conversion we cannot introduce references to free variables in arbitrary contexts. Since the normal η-equivalence for existential types introduces references to free variables in an arbitrary context, it is not valid in our closure-converted language.

7.2.1 Meta-Theory

As the target language is relatively standard, I do not prove any meta-theoretic results about CoC^{CC}; I sketch most of a model below. I conjecture that CoC^{CC} is consistent and type safe, strongly normalizing, and satisfies subject reduction. However, decidability is an open question; the presentation of translucent types is not syntax directed and likely requires changes to ensure decidability.

Both translucent types and existential types are easily modeled in CIC. Existential types are included in Coq's standard library, so I focus on translucent types. I give an implementation in Appendix J, but give an intuitive presentation here.

The translucent type is easily modeled with the identity type as follows.

$$\llbracket e \Rightarrow B \rrbracket^\circ \overset{\text{def}}{=} \Pi\, x : \llbracket A \rrbracket^\circ.\, x = \llbracket e \rrbracket^\circ \rightarrow \llbracket B \rrbracket^\circ$$
$$\text{where } e : A$$

Here, I write $\llbracket e \rrbracket^\circ$ to give a model of e in ECC^D extended with existential types and the identity type. The translucent type is modeled as a function that demands some argument x of type A and a proof that x is equal to e, using the identity type.

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

$$\begin{array}{rcl}
[\![x]\!] & \stackrel{\text{def}}{=} & \mathbf{x} \\
[\![\star]\!] & \stackrel{\text{def}}{=} & \star \\
[\![\Pi\, x : A.\, B]\!] & \stackrel{\text{def}}{=} & \exists\, (\alpha : \star, \mathbf{n} : \alpha).\, \mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \\
[\![\lambda\, x : A.\, e]\!] & \stackrel{\text{def}}{=} & \mathbf{pack}\, \langle \Sigma\, (\mathbf{x}_i : [\![A_i]\!] \ldots), \langle \mathbf{x}_i \ldots \rangle, \\
& & \qquad (\lambda\, (\mathbf{n} : \Sigma\, (\mathbf{x}_i : [\![A_i]\!] \ldots), \mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\, \mathbf{in}\, [\![A]\!]). \\
& & \qquad \quad \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\, \mathbf{in}\, [\![e]\!]) \rangle \\
& & x_i : A_i \ldots \; = \mathrm{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma) \\
[\![e_1\ e_2]\!] & \stackrel{\text{def}}{=} & \mathbf{unpack}\, \langle \alpha, \mathbf{n}, \mathbf{f} \rangle = [\![e_1]\!]\ \mathbf{in}\, \mathbf{f}\ \mathbf{n}\ \ [\![e_2]\!] \\
[\![\mathsf{let}\, x = e : A\, \mathsf{in}\, e']\!] & \stackrel{\text{def}}{=} & \mathbf{let}\, \mathbf{x} = [\![e]\!] : [\![A]\!]\ \mathbf{in}\ [\![e']\!] \\
& \vdots &
\end{array}$$

Figure 7.5: Parametric Closure Conversion from CoC^D to CoC^{CC} (excerpts)

The introduction and elimination forms are modeled as the following.

$$\begin{array}{rcl}
[\![\mathbf{e} : \mathbf{e}' \Rightarrow \mathbf{B}]\!]^\circ & \stackrel{\text{def}}{=} & \lambda\, (x : [\![\mathbf{A}]\!]^\circ, y : x = [\![\mathbf{e}']\!]^\circ).\, \mathsf{case}\ y\ \mathsf{of}\ \mathsf{refl}.\, [\![\mathbf{e}]\!]^\circ\ \ x \\
[\![\mathbf{e}\ \mathbf{e}']\!]^\circ & \stackrel{\text{def}}{=} & [\![\mathbf{e}]\!]^\circ\ \ [\![\mathbf{e}']\!]^\circ\, \mathsf{refl} \\
& & \text{where } \mathbf{e} : \mathbf{e}' \Rightarrow \mathbf{B}
\end{array}$$

The introduction form is modeled as a function of two arguments, which pattern matches on the equality proof and applies the underlying function $\mathbf{e}$ to the argument $\mathbf{x}$. The elimination form simply applies the translucent function $\mathbf{e}$ to its argument $\mathbf{e}'$ and the proof of equality refl.

The new η-equivalence for closures is the only non-standard rule and constructing a model of the rule requires a parametricity argument that I leave for future work. While essentially based on the η-principle for existential types, it is not exactly the same. It should be similar in some ways to the model for the CPS target language in Chapter 6. That is, a model should be possible using the parametricity translation into the extensional Calculus of Constructions. However, it is unclear how to formalize the relation between two environments of type $\boldsymbol{\alpha}$. Intuitively, we must allow any two environments, with potentially different sets of free variables, to be related as long as the codes are related after inlining the environments into the codes.

7.3 Parametric Closure Conversion Translation

In Figure 7.5, I present the key parametric closure-conversion rules. As in Chapter 5, formally, this translation must be defined on typing derivations, but for brevity I present the key translation rules on syntax. I define the complete translation in Appendix K

Figure K.3 and Figure K.4. The translation of dependent function types is the key to understanding our translation. As described in Section 7.1, the idea in this translation is to translate dependent functions of type $\Pi x : A.\, B$ to existential packages of type $\exists \alpha : \star, \mathbf{n} : \alpha.\, (\mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B})$. In the translation of types, we leave type variables free in the type of the closure–*i.e.*, in $\mathbf{A}$ and $\mathbf{B}$–but we leave them free under a translucent type $(\mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B})$ that describes the new closed Code. In the translation of functions, $[\![\lambda\, x : A.\, e]\!]$, we produce closed type annotations that are only valid with respect to the environment we produce in the closure. When we check that the closure produced by the translation has the type produced by the translation, *i.e.*, when showing type preservation for functions, the translucent type introduces an equality $\mathbf{n} = env$ into the type, unifying the closed type of the closure and the open type of the code.

The rest of the translation is straightforward. We translate each $\star$ to $\star$. Similarly, we translate names x into $\mathbf{x}$. All omitted rules are structural.

7.3.1 Type Preservation

In this section, I show that parametric closure-conversion is type preserving. I follow the standard architecture presented in Chapter 3.

Recall from Section 7.1 that the type preservation argument relies on an equivalence between a type $\mathbf{A}$ with free variables $\mathbf{x}_i \ldots$ and the same type with those free variables projected out of the closure's environment $\langle \mathbf{x}_i \ldots \rangle$, *i.e.*, $\mathbf{A} \equiv \mathbf{A}[\langle \mathbf{x}_i \ldots \rangle / \langle \mathbf{x}_i \ldots \rangle]$. Since the environment is always generated from the same free variables, this substitution is essentially a no-op. This argument is used in several proofs. I formalize this equivalence in Lemma 7.3.1. The proof is a straightforward computation.

Lemma 7.3.1. $\mathbf{\Gamma}, \mathbf{n} = \langle \mathbf{x}_i \ldots \rangle : \mathbf{A}' \vdash (\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e}) \equiv \mathbf{e}$.

Proof. $\mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\ \mathbf{in}\ \mathbf{e} \rhd_\delta \mathbf{let}\ \langle \mathbf{x}_i \ldots \rangle = \langle \mathbf{x}_i \ldots \rangle\ \mathbf{in}\ \mathbf{e} \rhd_\zeta^n \mathbf{e}[\mathbf{x}_i / \mathbf{x}_i] = \mathbf{e}$ □

Next I prove compositionality. As in Chapter 5, this is where we use most of the complexity of the target language, such as translucent types and the η-principle for closures. I go into detail for the key proof cases, *i.e.*, those for the encoding of closures.

Lemma 7.3.2 (Compositionality). $[\![(e[e'/x])]\!] \equiv [\![e]\!][[\![e']\!]/x]$

Proof. By cases on structure of e. Omitted cases are straightforward.

Case: $e = \Pi\, y : A.\, B$

Show $[\![((\Pi\, y : A.\, B)[e'/x])]\!] \equiv [\![(\Pi\, y : A.\, B)]\!][[\![e']\!]/x]$.

$$
\begin{aligned}
& [\![((\Pi\, y : A.\, B)[e'/x])]\!] \\
= \;& [\![(\Pi\, y : A[e'/x].\, B[e'/x])]\!] && (141)\\
& \text{by substitution} \\
= \;& \exists\,(\alpha : \star, \mathbf{n} : \alpha).\, \mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{y} : [\![A]\!][[\![e']\!]/\mathbf{x}].\, [\![B]\!][[\![e']\!]/\mathbf{x}] && (142)\\
& \text{by definition of translation} \\
= \;& (\exists\,(\alpha : \star, \mathbf{n} : \alpha).\, \mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{y} : [\![A]\!]\,.\, [\![B]\!])[[\![e']\!]/x] && (143)\\
& \text{by substitution} \\
= \;& [\![(\Pi\, y : A.\, B)]\!][[\![e']\!]/x] && (144)
\end{aligned}
$$

Case: $e = \lambda\, x : A.\, e_1$.

This case essentially asks "when are two closures equivalent?", since substitution on a function changes its closure by changing the environment. The η rules for closures allows us to answer that question. When substituting into a closure before translation, we produce a closure whose environment contains only free variables. When substituting into a closure after translation, we can put a (potentially open) term into the environment of a closure.

$$
\begin{aligned}
p_L = [\![(e[e'/x])]\!] = \;& \mathbf{pack}\,\langle \Sigma'_{env}, env', \boldsymbol{\lambda}\,\mathbf{n} : \Sigma'_{env}, \mathbf{y} : \mathbf{A}_L. \\
& \qquad \mathbf{let}\, env' = \mathbf{n}\,\mathbf{in}\, [\![e_1]\!][[\![e']\!]/\mathbf{x}]\rangle \\
\text{where } env' = \;& \langle \mathbf{x}_0, \dots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_m, \mathbf{x}_{m+1}, \dots, \mathbf{x}_n \rangle \\
& \mathbf{x}_0, \dots, \mathbf{x}_{i-1}, \mathbf{x}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_m = \mathrm{fv}([\![e]\!]) \\
& \mathbf{x}_{m+1}, \dots, \mathbf{x}_n = \mathrm{fv}([\![e']\!]) \\
& \mathbf{A}_L = \mathbf{let}\, env' = \mathbf{n}\,\mathbf{in}\, [\![A]\!] \\
p_R = [\![e]\!][[\![e']\!]/x] = \;& \mathbf{pack}\,\langle \Sigma_{env}, env, \boldsymbol{\lambda}\,\mathbf{n} : \Sigma_{env}, \mathbf{y} : \mathbf{A}_R.\, \mathbf{let}\, env = \mathbf{n}\,\mathbf{in}\, [\![e]\!]\rangle \\
\text{where } env = \;& \langle \mathbf{x}_0, \dots, \mathbf{x}_{i-1}, [\![e']\!], \mathbf{x}_{i+1}, \dots, \mathbf{x}_m \rangle \\
& \mathbf{x}_0, \dots, \mathbf{x}_{i-1}, \mathbf{x}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_m = \mathrm{fv}([\![t]\!]) \\
& \mathbf{t}_R = \mathbf{let}\, env = \mathbf{n}\,\mathbf{in}\, [\![A]\!]
\end{aligned}
$$

Note that in p_L, the environment env' is missing $\mathbf{x}$ between $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$ and contains the free variables from $[\![e']\!]$. In p_R, the environment env contains $[\![e']\!]$ in place of $\mathbf{x}$. To show that $p_L \equiv p_R$, it suffices by Rule CC-$\equiv$-η_1 to show $[\![e]\!][[\![e']\!]/\mathbf{x}] \equiv \mathbf{unpack}\,\langle \alpha, \mathbf{n}, \mathbf{f}\rangle = p_R\,\mathbf{in}\,\mathbf{f}\ \mathbf{n}\ \mathbf{y}$. Observe that

$$
\begin{aligned}
& \mathbf{unpack}\,\langle \alpha, \mathbf{n}, \mathbf{f}\rangle = p_R\,\mathbf{in}\,\mathbf{f}\ \mathbf{n}\ \mathbf{y} \\
& \triangleright^*\ \mathbf{let}\, env = env\,\mathbf{in}\, [\![e]\!] \\
& \triangleright_\zeta\ \mathbf{let}\,\langle \mathbf{x}_0, \dots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_m \rangle = \langle \mathbf{x}_0, \dots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_m \rangle\ \mathbf{in}\, [\![e]\!][[\![e']\!]/\mathbf{x}] \\
& \equiv [\![e]\!][[\![e']\!]/\mathbf{x}] \text{ by Lemma 7.3.1.}
\end{aligned}
$$

□

Now we can show that the translation preserves reduction up to equivalence, *i.e.*, that the translation $\mathbf{e}$ of a term e weakly simulates one step of reduction of e. Since Lemma 7.3.2 is in terms of definitional equivalence, I show weak simulation up to definitional equivalence. I give the key cases.

Lemma 7.3.3 (Preservation of Reduction)**.** *If* $\Gamma \vdash \mathsf{e} \rhd_x \mathsf{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \rhd^* \mathbf{e}$ *and* $\mathbf{e} \equiv [\![\mathsf{e}']\!]$

Proof. By cases on the judgment $\mathsf{e} \rhd_x \mathsf{e}'$. All cases are straightforward by computation and Lemma 7.3.2.

Case: $\mathsf{x} \rhd_\delta \mathsf{e}'$ where $\mathsf{x} = \mathsf{e}' \in \Gamma$

Suffices to show that $\mathbf{x} \rhd_\delta [\![\mathsf{e}']\!]$ where $\mathbf{x} = [\![\mathsf{e}']\!] \in \Gamma$, which follows by definition of $\vdash \Gamma \rightsquigarrow \mathbf{\Gamma}$.

Case: $(\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}_1)\ \mathsf{e}_2 \rhd_\beta \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]$

We must show that $[\![((\lambda\, \mathsf{x} : \mathsf{A}.\, \mathsf{e}_1)\ \mathsf{e}_2)]\!] \rhd^* \mathbf{e}' \equiv [\![(\mathsf{e}_1[\mathsf{e}_2/\mathsf{x}])]\!]$.

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

$$= \mathbf{unpack}\ \langle \boldsymbol{\alpha}, \mathbf{n}, \mathbf{f}\rangle = (\mathbf{pack}\ \langle \mathbf{A}', \langle \mathbf{x}_i \ldots\rangle, \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\ \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{e}_1]\!]\rangle)\ \mathbf{in}\ \mathbf{f}\ \mathbf{n}\ \ [\![\mathsf{e}_2]\!] \tag{145}$$

$$\rhd_\exists\ (\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\, \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \mathbf{n}\ \mathbf{in}\ [\![\mathsf{e}_1]\!])\ \langle \mathbf{x}_i \ldots\rangle\ \ [\![\mathsf{e}_2]\!] \tag{146}$$

$$\rhd_\beta\ \mathbf{let}\ \langle \mathbf{x}_i \ldots\rangle = \langle \mathbf{x}_i \ldots\rangle\ \mathbf{in}\ [\![\mathsf{e}_1]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}] \tag{147}$$

$$\equiv\ [\![\mathsf{e}_1]\!][[\![\mathsf{e}_2]\!]/\mathbf{x}] \tag{148}$$

by Lemma 7.3.1

$$\equiv\ [\![(\mathsf{e}_1[\mathsf{e}_2/\mathsf{x}])]\!] \tag{149}$$

by Lemma 7.3.2 (Compositionality)

□

Lemma 7.3.4 (Preservation of Conversion)**.** *If* $\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'$ *then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \rhd^* \mathbf{e}$ *and* $[\![\Gamma]\!] \vdash \mathbf{e} \equiv [\![\mathsf{e}']\!]$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash \mathsf{e} \rhd^* \mathsf{e}'$. The proof is entirely uninteresting, following essentially from Lemma 7.3.3 (Preservation of Reduction). □

To show that equivalence is preserved, we require η-equivalence for closures in the case of η-equivalent source terms.

Lemma 7.3.5 (Preservation of Equivalence)**.** *If* $\Gamma \vdash \mathsf{e} \equiv \mathsf{e}'$*, then* $[\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \equiv \mathsf{e}'^{+}$.

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

Case: Rule $\equiv$ By assumption, $e \rhd^* e_1$ and $e' \rhd^* e_1$.

By Lemma 7.3.4, $[\![e]\!] \rhd^* \mathbf{e}$ and $\mathbf{e} \equiv [\![e_1]\!]$, and similarly. $[\![e']\!] \rhd^* \mathbf{e}'$ and $\mathbf{e}' \equiv [\![e_1]\!]$. The result follows by symmetry and transitivity of $\equiv$.

Case: Rule $\equiv$-η_1 By assumption, $e \rhd^* \lambda\, x : t.\, e_1$, $e' \rhd^* e_2$ and $e_1 \equiv e_2\ x$.

Must show $[\![e]\!] \equiv [\![e']\!]$.

By Lemma 7.3.4 (Preservation of Conversion), $[\![e]\!] \rhd^* \mathbf{e}$ and $\mathbf{e} \equiv [\![\lambda\, x : t.\, e_1]\!]$, and similarly $[\![e']\!] \rhd^* \mathbf{e}'$ and $\mathbf{e}' \equiv [\![e_2]\!]$.

By transitivity, it suffices to show $[\![\lambda\, x : t.\, e_1]\!] \equiv [\![e_2]\!]$.

By Rule $\equiv$-η_1 and Lemma 7.3.1, it suffices to show

$$[\![e_1]\!] \equiv \mathbf{unpack}\ \langle \boldsymbol{\alpha}, \mathbf{n}, \mathbf{f} \rangle = [\![e_2]\!]\ \mathbf{in\, f\ n\ x}.$$

Note that $\mathbf{unpack}\ \langle \boldsymbol{\alpha}, \mathbf{n}, \mathbf{f} \rangle = [\![e_2]\!]\ \mathbf{in\, f\ n\ x}$ is exactly $[\![(e_2\ x)]\!]$, so the result follows by the induction hypothesis.

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

□

Now we can prove the central lemma necessary for showing type preservation.

Lemma 7.3.6 (Type and Well-formedness Preservation)**.**

1. *If* $\Gamma \vdash$ *then* $\vdash [\![\Gamma]\!]$
2. *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$

Proof. Parts 1 and 2 proven by simultaneous induction on the mutually defined judgments $\Gamma \vdash$ and $\Gamma \vdash e : A$.

Part 1 follows easily by induction and part 2. I give the key cases for part 2.

Case: Rule *, follows by part 1.

Case: Rule Var, follows by part 1.

Case: Rule Pi-*

We have that $\Gamma \vdash \Pi\, x : A.\, B : \star$.

We must show that $\boldsymbol{\Gamma} \vdash \exists\, (\boldsymbol{\alpha} : \star, \mathbf{n} : \boldsymbol{\alpha}).\, \mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{x} : [\![A]\!]\, .\, [\![B]\!] : \star$

By Rule Exist, it suffices to show $\boldsymbol{\Gamma}, \boldsymbol{\alpha} : \star, \mathbf{n} : \boldsymbol{\alpha} \vdash (\mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{x} : [\![A]\!]\, .\, [\![B]\!]) : \star$.

By Rule $\Rightarrow$, it suffices to show $\boldsymbol{\Gamma}, \boldsymbol{\alpha} : \star, \mathbf{n} : \boldsymbol{\alpha} \vdash \mathbf{Code}\, \mathbf{n}' : \boldsymbol{\alpha}, \mathbf{x} : [\![A]\!]\, .\, [\![B]\!] : \star$

By Rule Code-*, it suffices to show $\boldsymbol{\Gamma}, \boldsymbol{\alpha} : \star, \mathbf{n} : \boldsymbol{\alpha}, \mathbf{n}' : \boldsymbol{\alpha}, \mathbf{x} : [\![A]\!] \vdash [\![B]\!] : \star$, which follows by the induction hypothesis applied to $\Gamma, x : A \vdash B : \star$.

Case: Rule Lam

We must show that $\Gamma \vdash [\![(\lambda\, x : A.\, e)]\!] : [\![(\Pi\, x : A.\, B)]\!]$

The key to this proof is to show that

$\Gamma \vdash (\lambda\, (n : A', x : \text{let}\, env = n\, \text{in}\, [\![A]\!]).\, \text{let}\, env = n\, \text{in}\, [\![e]\!]) : env \Rightarrow \text{Code}\, x : [\![A]\!]\,.\, [\![B]\!]$

where $env = \langle x_0, \cdots x_n \rangle$
$A' = \Sigma\, (x_0 : A_0, \ldots, x_n : A_n)$

Note that by definition, $\Gamma \vdash env : A'$.

Note also that $\Gamma, n = env : A' \vdash \text{let}\, env = n\, \text{in}\, [\![A]\!] \equiv [\![A]\!]$, by Lemma 7.3.1.

So, by Rule Conv and Rule $\Rightarrow$, it suffices to show:

$\Gamma \vdash \lambda\, (n : A', x : \text{let}\, env = n\, \text{in}\, [\![A]\!]).\, \text{let}\, env = n\, \text{in}\, [\![e]\!] :$
$\text{Code}\, (n : A', x : \text{let}\, env = n\, \text{in}\, [\![A]\!]).\, \text{let}\, env = n\, \text{in}\, [\![B]\!]$

By Rule Code and Rule Let, it suffices to show that

$n : A', x : \text{let}\, env = n\, \text{in}\, [\![A]\!]\,, x_0 = \pi_0\, n : t_0, \ldots, x_n = \pi_n\, n : A_n \vdash [\![e]\!] : [\![B]\!]$

Note that by the induction hypothesis we know that $\Gamma, x : [\![A]\!] \vdash [\![e]\!] : [\![B]\!]$.

The goal follows since, by definition of the translation, the free variables of $[\![e]\!]$ are exactly x_0, x_n, and all other variables of Γ are not referenced in $[\![e]\!]$ or $[\![B]\!]$.

Case: Rule App

We must show that $\Gamma \vdash [\![(e\ e')]\!] : [\![(B[e'/x])]\!]$

That is, we must show

$\Gamma \vdash \text{unpack}\, \langle \alpha, n, f \rangle = [\![e]\!]\ \text{in}\, f\ n\ \ [\![e']\!] : [\![(B[e'/x])]\!]$

By Rule Unpack, it suffices to show, (1):

$\Gamma \vdash [\![e]\!] : \exists\, \alpha : \star, n : \alpha.\, n \Rightarrow \text{Code}\, x : [\![A]\!]\,.\, [\![B]\!],$

which follows immediately from the induction hypothesis, and (2):

$\Gamma, \alpha : \star, n : \alpha, f : n \Rightarrow \text{Code}\, x : [\![A]\!]\,.\, [\![B]\!] \vdash f\ n\ \ [\![e']\!] : [\![(B[e'/x])]\!]$

By Rule Conv and Lemma 7.3.2 (Compositionality), it suffices to show

$\Gamma, \alpha : \star, n : \alpha, f : n \Rightarrow \text{Code}\, x : [\![A]\!]\,.\, [\![B]\!] \vdash f\ n\ \ [\![e']\!] : [\![B]\!][[\![e']\!]/x]$

By Rule TrApp, it suffices to show that

$\Gamma, \alpha : \star, n : \alpha, y : n \Rightarrow \text{Code}\, x : [\![A]\!]\,.\, [\![B]\!] \vdash [\![e']\!] : [\![A]\!]$, which follows by the induction hypothesis.

□

Corollary 7.3.7 (Type Preservation)**.** *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![A]\!]$.

7.3.2 Compiler Correctness

The proof of correctness with respect to separate compilation follows exactly the pattern in Chapter 3. I define linking by substitution, as usual, and define observations and whole program as evaluation to booleans.

Theorem 7.3.8 (Separate Compilation Correctness). *If* $\Gamma \vdash \mathsf{e}$, $\Gamma \vdash \gamma$, $[\![\Gamma]\!] \vdash \boldsymbol{\gamma}$, *and* $[\![\gamma]\!] \equiv \boldsymbol{\gamma}$ *then* $\mathsf{eval}(\gamma(\mathsf{e})) \approx \mathbf{eval}(\boldsymbol{\gamma}([\![\mathsf{e}]\!]))$

Proof. Since $\equiv$ on observations implies $\approx$, it suffices to show $\mathsf{eval}(\gamma(\mathsf{e})) \equiv \mathbf{eval}(\boldsymbol{\gamma}([\![\mathsf{e}]\!]))$. Since the translation commutes with substitution, preserves equivalence, reduction implies equivalence, and equivalence is transitive, the following diagram commutes.

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

□

Corollary 7.3.9 (Whole-Program Correctness). *If* $\cdot \vdash \mathsf{e}$ *then* $\mathsf{eval}(\mathsf{e}) \approx \mathbf{eval}([\![\mathsf{e}]\!])$

8 CONCLUSIONS

In this chapter, I summarize the lessons of the four translations presented earlier in this dissertation and conjecture how to apply those lessons to complete the project described in Chapter 1. In a sense, I consider the previous four chapters to be mathematical case studies. When this project began, I had no idea how hard any given type-preserving translation would be. The only prior work on dependent-type-preserving compilation for full-spectrum dependent types studied CPS, which proved difficult. I approached my thesis by attempting the standard type-preserving translations to see what worked and what failed. From these case studies, I conclude lessons about the individual translations, and general lessons about dependent-type preservation. I end by conjecturing how to apply these lessons to future work: completing the compiler from Coq to dependently typed assembly.

8.1 Viability of the Individual Translations

Recall from Chapter 1 my stated thesis: type-preserving compilation of dependent types is a theoretically viable technique for eliminating miscompilation errors and linking errors. The key word in this thesis is *viable*, that is, "capable of working successfully; feasible". For type-preserving compilation to be, theoretically, capable of working successfully for eliminating errors, we must be able to apply the theory to real tools and languages used in practice. I summarize the "viability" of the translations in Figure 8.1—describing which translations are most promising in the sense that they should scale to use in practice, and which need further work.

Two of the translations—the ANF translation, Chapter 4, and the abstract closure conversion, Chapter 5—should scale to languages used in practice, such as Coq. They support all the core features of dependency and are orthogonal to additional axioms such as parametricity and impredicativity, meaning they should scale to a variety of dependently typed languages used in practice. I consider these two the most promising for further development into a practical compiler for Coq.

The other two translations—the CPS translation, Chapter 6, and the parametric closure conversion, Chapter 7—do not scale to Coq in their current form, although I conjecture that we can apply more advanced type theories to the target language to develop variants of these translations that do scale. In their current form, both these translations rely on parametricity and impredicativity for type preservation. The CPS translation relies on parametricity and impredicativity of all computationally relevant functions, since it encodes computations using parametric functions, and the type of

ANF (Chapter 4)	CPS (Chapter 6)
Viable: works with higher universes; orthogonal to parametricity and impredicativity.	Not Yet Viable: unknown how to scale to higher universes; currently requires parametricity and impredicativity of all computationally relevant functions.
Abstract CC (Chapter 5)	Parametric CC (Chapter 7)
Viable: works with higher universes; orthogonal to parametricity and impredicativity.	Not Yet Viable: unknown how to scale to higher universes; currently requires parametricity and impredicativity of all computationally relevant recursive closures.

Figure 8.1: Viability of Translations

computations require impredicativity. Similarly, the parametric closure type requires parametricity to justify the η-equivalence for closures, and would require impredicativity to express recursive closures. Unfortunately, using multiple impredicative universes in the same hierarchy is inconsistent in general, so we can't create a different impredicative universe for each computationally relevant universe in the source. Furthermore, the combination of impredicativity and computational relevance is inconsistent with some set-theoretic axioms that are used in Coq in practice; and parametricity is orthogonal to dependent type theory, so requiring it disallows some axioms that user programs or proofs could soundly rely on.

However, I conjecture that both of these translations could be extended to scale to Coq by using more advanced type theories in the target language. For example, we could introduce a separate type for computations, instead of encoding them using the same function type that source functions are compiled to. It might be possible to safely allow impredicativity in this computation type, without requiring impredicativity in all functions. Unfortunately, I'm unfamiliar with any type theories that mix predicative and impredicative quantification at every universe level, as this idea would require. Similarly, we only require parametricity for computations and closures. If we could isolate parametricity in a modality, we might be able to use it without requiring it for all functions or closures. Nuyts and Devriese (2018) develop a type theory with a modality for parametricity, but it does not yet support higher universes.

8.2 Lessons for Dependent-Type Preservation

In Chapter 3, I describe the general proof recipe I use to prove dependent-type preservation, but there's a missing step before we can use that proof recipe: model the dependencies of the implicit semantic object from the high-level source as an explicit syntactic object in the low-level target. For example, CPS and ANF are encoding continuations in the syntax of the target, and closure conversion is encoding closures.

In each of these translations, I had to come up with a typed encoding that preserved all the dependencies from the source language. This is the hard part in each of the translations I presented, but there is a pattern to come up with this encoding, which I describe in this section.

THE PATTERN

1. Model the semantic object as syntax.

   For example, CPS and ANF model continuations. In the source language, continuations are implicit, and the semantics ensure they are used linearly. When we model continuations explicitly in the syntax, we must preserve this property; trying to preserve dependent types essentially forces us to. The CPS translation modeled continuations by relying on parametricity, which enforces linearity as long as there are no effects. ANF modeled this by keeping continuations non-first-class, enforcing linearity in the syntax.

   Closure conversion models closures. In the source closures have the following properties: a closure's environment is not part of the type, types can refer to the environment, and no term can modify a closure's environment. Both the abstract closure and the closure modeled with existential types, via parametricity, enforce all of these properties.

2. Record machine steps in the typing derivation.

   In a source dependently typed language, we compose by nesting expressions and by substitution. To compose an expression $\mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}. \mathsf{B}$ with a second projection snd, we just nest them as in $\mathsf{snd}\, \mathsf{e}$. This shows up in the typing derivation in dependent typing rules like Rule App and Rule Snd. For example, Rule Snd is reproduced below.

   $$\begin{tikzcd}[column sep=tiny] \frac{\Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}. \mathsf{B}}{\Gamma \vdash \mathsf{snd}\, \mathsf{e} : \mathsf{B}[\mathsf{fst}\, \mathsf{e}/\mathsf{x}]} \end{tikzcd}$$

   Note how the expression e is copied into the typing derivation, using substitution to compose the expressions B and $\mathsf{fst}\, \mathsf{e}$.

   In a target language, we move away from a compositional expression-based syntax and instead encode the steps of a machine (abstract or concrete) directly in syntax. In dependently typed languages, machine steps affect typing. For example, the expression $\mathsf{snd}\, \mathsf{e}$ in CPS is represented (roughly) as $\mathsf{e}^{\div}\ (\boldsymbol{\lambda}\, \mathbf{y}. \mathbf{k}\ (\mathbf{snd}\, \mathbf{y}))$, where $\mathbf{k}$ is the current continuation. This represents three steps of computation, and should be read as "the machine first evaluates $\mathsf{e}^{\div}$, and then evaluates $\mathbf{snd}\, \mathbf{y}$, and then continues the computation at $\mathbf{k}$".

In a dependently typed language, we need to recover dependencies on expressions that have been decomposed into steps in a machine. For example, naïvely typing the above CPS expression gives us, roughly, the following typing derivation.

$$\dfrac{\dfrac{}{\Gamma \vdash \mathsf{e}^{\div} : \mathbf{Cont} \to \mathbf{R}} \qquad \dfrac{\Gamma, \mathbf{y} : \mathbf{\Sigma\, x} : \mathsf{A}^{+}. \mathsf{B}^{+} \vdash \mathbf{snd\, y} : \mathsf{B}^{+}[\mathbf{fst\, y}/\mathbf{x}]}{\vdots}}{\Gamma \vdash \mathsf{e}^{\div}\ (\boldsymbol{\lambda}\, \mathbf{y}. \mathbf{k}\ (\mathbf{snd\, y})) : \mathbf{R}}$$

Here, I ignore the types of the result $\mathbf{R}$ and the continuation $\mathbf{Cont}$. Recall from Chapter 6 that this derivation fails. In the source, $\mathsf{snd\, e} : \mathsf{B}[\mathsf{fst\, e}/\mathsf{x}]$, that is, $\mathsf{snd\, e}$ depends on e. However, after decomposing the expression $\mathsf{snd\, e}$ into three machine steps, we end up type checking $\mathbf{snd\, y}$, forgetting the dependency on $\mathsf{e}^{\div}$. To fix this, we have to record all the machine steps that lead up to $\mathbf{snd\, y}$ so we can use them to re-establish the dependency on the computations $\mathsf{e}^{\div}$. This is essentially what both the CPS translation in Chapter 6 and the ANF translation in Chapter 4: record machine steps in typing derivations.

Intuitively, the CPS typing derivation should look like the following.

$$\dfrac{\dfrac{}{\Gamma \vdash \mathsf{e}^{\div} : \mathbf{Cont} \to \mathbf{R}} \qquad \dfrac{\Gamma, \mathbf{y} : \mathbf{\Sigma\, x} : \mathsf{A}^{+}. \mathsf{B}^{+}, \mathbf{y} := \mathsf{e}^{\div} \vdash \mathbf{snd\, y} : \mathsf{B}^{+}[\mathbf{fst\, y}/\mathbf{x}]}{\Gamma, \mathbf{y} : \mathbf{\Sigma\, x} : \mathsf{A}^{+}. \mathsf{B}^{+}, \mathbf{y} := \mathsf{e}^{\div} \vdash \mathbf{k}\ (\mathbf{snd\, y}) : \mathbf{R}}}{\Gamma \vdash \mathsf{e}^{\div}\ (\boldsymbol{\lambda}\, \mathbf{y}. \mathbf{k}\ (\mathbf{snd\, y})) : \mathbf{R}}$$

That is, when go up the derivation, we record the machine step $\mathbf{y} := \mathsf{e}^{\div}$. We should read this as "the machine step $\mathbf{snd\, y}$ takes place after the machine steps of $\mathsf{e}^{\div}$, which leave the value of those steps in $\mathbf{y}$". This records a dependency on $\mathsf{e}^{\div}$, allowing us to re-establish the dependency from the source language.

Both CPS and ANF record machine steps going up the typing derivation, pushing machine steps into the leaves. This happens since CPS and ANF un-nest and sequence expressions. In Chapter 6, we encoded this machine step as a definition $\mathbf{y} = \mathsf{e}^{\div}\ \mathbf{id}$, that is, as the computation applied to the halt continuation. In Chapter 4, we encoded machine steps as the series of definitions introduced by $\mathbf{let}$ bindings. By recording the machine steps in the typing derivation, we can recover dependencies that have been separated from a single expression into multiple machine steps.

Closure conversion is simpler, since it does not sequence expressions and thus does not need to push dependencies into the leaves of the derivation trees. However, closure conversion still introduces a machine step, in particular, creating a closure. The typing rule for creating a closure records a machine step in the typing derivation by substituting the closure's environment into the closure's type. Recall the typing rule for closures, Rule Clo, from Chapter 5, reproduced below.

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

Closure conversion introduces an explicit abstraction over the environment $\mathbf{n}$. Creating a closure is the machine step that instantiates the environment. For dependent types, the machine step of creating a closure must re-introduce dependencies in the type, by substituting the environment $\mathbf{e}'$ for the environment variable $\mathbf{n}$. With abstract closure conversion, Chapter 5, we model this directly with a new syntactic form. With parametric closure conversion, Chapter 7, we model this by encoding the machine step using singleton types.

3. Interpret machine steps in definitional equivalence.

   Definitional equivalence is key to deciding type equivalence, but after expressions change into machine steps, we must interpret machine steps instead of just normalizing expressions. For example, in CPS we record the machine step $\mathbf{y} := \mathsf{e}^{\div}$ with the particular encoding $\mathbf{y} = \mathsf{e}^{\div}\ \mathbf{id}$. The reason for this encoding, and for the equivalence rule Rule T-Cont, reproduced below, is to interpret machine steps by turning CPS'd computations back into expressions that we can simply normalize.

$$\frac{}{\Gamma \vdash \mathbf{e}\ \mathbf{A}\ (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{B}.\,\mathbf{e}') \equiv (\boldsymbol{\lambda}\,\mathbf{x} : \mathbf{e}'.\,)\ (\mathbf{e}\ \mathbf{B}\ \mathbf{id})}$$

   Rule T-Cont allows rewriting any CPS'd expression into an expression representation of a machine step, so we can normalize machine steps the way we normalize expressions in the source language. A CPS'd expression applied to the halt continuation simply runs to a value, giving us an interpretation of machine steps as expressions. In ANF, the lemma Lemma 4.2.5 corresponds to the interpretation of machine steps. It tells us we can interpret the sequences of definitions given by an ANF term as the original expression. In closure conversion, the η-equivalence rule for closures gives us an interpretation of machine steps. Creating a closure is a machine step, and we interpret them as expressions by simply inlining the environment.

In each of the above steps, we have to consider two things: whether the encoding or interpretation requires additional axioms, or whether they reinterpret existing types. The reason that CPS and parametric closure conversion are problematic is that their encodings requires parametricity and impredicativity in order to model the machine objects and interpret machine steps. This would not be a big problem if the new axioms were somehow isolated from user code that could rely on conflicting axioms. The CPS translation also makes the mistake of reusing (and thus, reinterpreting) an existing type: the function type. This means that the new axioms are now required of pre-existing

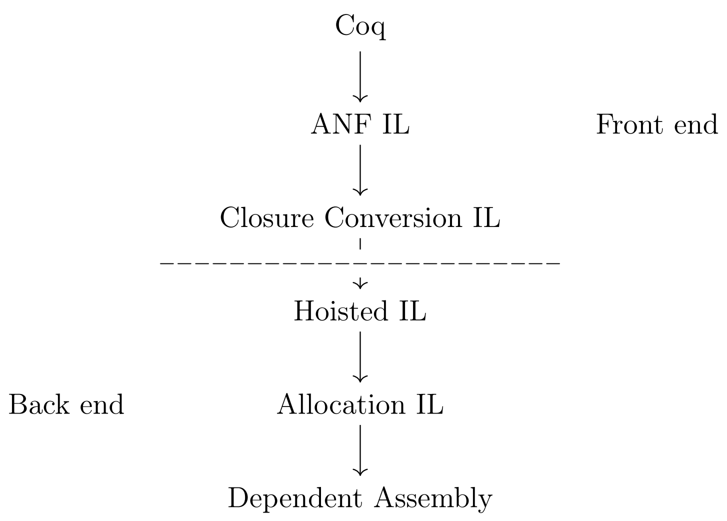

Figure 8.2: Future Type-Preserving Compiler

types, *i.e.*, the function type is reinterpreted. This is a problem when we're trying to keep the translation as broadly applicable as possible. By contrast, the ANF and abstract closure conversion do not introduce new axioms, and the encodings in those translations do not reinterpret existing types. ANF does not introduce any new types at all, instead encoding machine steps in syntax. Abstract closure conversion introduces a new code type, and ensures that the dependent function type maintains its previous interpretation.

8.3 Future Work

This section is dedicated to Amal and Stephanie.

In this dissertation, I have only described translations in the front-end of the compiler—half of the compiler necessary to reach the dependently typed assembly language described in Chapter 1. I have also ignored many related practical issues, such as computational relevance, code size of assembly with type and proof annotations, or linking with untyped and/or effectful code. In this section, I explain how the lessons just described will help us finish the compiler. I then speculate about related practical issues.

8.3.1 To Dependently Typed Assembly

Recall that the standard model type preserving compiler, Figure 3.1 from Chapter 3, has five passes. Following this model, we want to eventually build the compiler described by Figure 8.2. This figure replaces CPS from the model with ANF, following the lessons discussed earlier in this chapter. We still need three more translations: hoisting, allocation, and code generation. How do we get from here to there? The pattern described in the previous section informs us how to begin designing these three translations.

HOISTING

Typographical Note. *In this section,* ECC^{CC} *is the source language and* $\mathit{ECC}^{CC/H}$*, a target language that I conjecture will be a syntactically restricted version of* ECC^{CC} *but whose formal definition I leave for future work, is the* ***target language****.*

Hoisting will be a trivially type-preserving translation, and can be phrased as an intra-language transformation in ECC^{CC} (the closure conversion IL). It will require no new features and can be encoded using **let** and definitions. This is because, unlike ANF, CPS, and closure conversion, hoisting does not model a new semantic object; it is a syntactic change to accommodate a later pass that will introduce a new object (the heap).

In this translation, we will lift all function definitions to be defined at the top-level. The next translation, explicit allocation, can then easily move functions from top-level definitions to heap-allocated code (labeled blocks). In general, hoisting is easy since closure conversion has already closed all functions, so they can be freely moved into a different scope.

Dependent types introduce a minor complication: functions defined earlier can appear in the types of closures defined later. For instance, suppose we have a source function $\mathsf{f} : \mathsf{Code}\,(\mathsf{n} : \mathsf{A}_1', \mathsf{x} : \mathsf{A}_1).\,\mathsf{B}_1$ and a closure $\mathsf{g} : \Pi\,\mathsf{x} : \mathsf{A}_2.\,\mathsf{B}_2$, where f is defined before g. Due to dependency, f (as a literal value, or by name) could appear in the type of g, such as $\mathsf{g} : \Pi\,\mathsf{x} : ((\langle\!\langle \mathsf{f}, \mathsf{n}' \rangle\!\rangle)\ \mathsf{e}').\,\mathsf{B}_2'$ (where n' is the environment for f) if f is a type constructor, or $\mathsf{g} : \Pi\,\mathsf{x} : (\mathsf{IsCorrect}\ \mathsf{f}).\,\mathsf{B}_2'$ if g requires a correctness proof for f before executing. This means when deciding type equivalence after hoisting, we will need to inline the definition of $\mathbf{f}$ into types. This is not a major complication; we can model top-level definitions using dependent **let** and *definitions*. Since definitions can be inlined during type equivalence, type equivalence should be preserved easily.

The syntax of programs after translation will look like the following,

$$\begin{array}{l}
\mathbf{let}\ \mathbf{f}_0 = \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}_0{\prime}, \mathbf{x}_0 : \mathbf{A}_0).\,\mathbf{e}_0 \\
\quad\vdots \\
\quad \mathbf{f}_n = \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}_n{\prime}, \mathbf{x}_n : \mathbf{A}_n).\,\mathbf{e}_n \\
\mathbf{in}\,\mathbf{e}
\end{array}$$

where there are no function literals in any of $\mathbf{A}_0{\prime}, ..., \mathbf{A}_n{\prime}, \mathbf{A}_0, ..., \mathbf{A}_n$ or $\mathbf{e}$. The hoisting translation must propagate function bindings out to the top-level, similar to how the ANF translation propagates intermediate computations out.

EXPLICIT ALLOCATION

Typographical Note. *In this section,* $\mathit{ECC}^{CC/H}$ *is the source language and* ECC^{H}*, a target language that I conjecture will be like* ECC^{CC} *but with a model of the heap and* *explicit allocation and dereference but whose formal definition I leave for future work, is the* ***target language****.*

Explicit allocation will be the primary challenge in the rest of the compiler. Like the previous passes, explicit allocation introduces a new semantic object, the heap, and new machine steps, allocation and dereference. Modeling these will require keeping track of the *definitions* of heap-allocated values during type checking; prior work only required the *types* of heap-allocated values during type checking (Morrisett et al., 1999). We may also require some new mechanism to prevent cycles in the heap.

The main problem for a dependent-type-preserving explicit-allocation pass is that the heap is necessarily unordered, which can admit cycles and thus inconsistency. A computation may jump to a function allocated early in the heap, but require a reference to a pair defined later in the heap. For example, suppose we have the following term.

$$\mathbf{\Gamma}; (\mathbf{l_1} := (\boldsymbol{\lambda}\,(\mathbf{n} : \&\mathbf{A'}, \mathbf{x} : \mathbf{A}).\,\mathbf{e}), \mathbf{l_2} := \mathbf{null} : \mathbf{A'}) \vdash \mathbf{set}\,\mathbf{l_2} = \mathbf{e}_{env}; (\mathbf{deref}\,\mathbf{l_1})\ \mathbf{l_2}\ \mathbf{e'} : \mathbf{B}$$

This term is calling the function at location $\mathbf{l_1}$. That function expects a reference ($\&$) to its environment of type $\mathbf{A'}$, and its usual argument of type $\mathbf{A}$. To call the function, we first set the environment in location $\mathbf{l_2}$, then dereference and call $\mathbf{l_1}$ with the environment $\mathbf{l_2}$ and some argument $\mathbf{e'}$ of type $\mathbf{A}$. This is a dependent function, so we need to compute the type $\mathbf{B}$ by instantiating the type of $\mathbf{e}$ with values from the *new* value of $\mathbf{l_2}$, after machine step $\mathbf{set}\,\mathbf{l_2}$.

We must allow this to model the heap correctly, but we must also prevent cycles in the heap to maintain logical consistency. It is well-known that introducing a model of the heap like the above into a terminating typed language can enable encoding nontermination (Landin's Knot). Since our dependently typed programs can represent proofs, this would result in inconsistency.

Prior work uses linear types to solve a similar problem, but it's unclear if this approach will scale to dependent types. Ahmed et al. (2007) allow cycles in the heap but rely on linear capabilities to still guarantee termination. Unfortunately, linear types do not integrate well with dependent types, since a dependent type requires references to a term during type checking, and references during run time. Both the type system and the run-time term will require a reference to the same object, but linear typing will restrict us to only one alias: we can either use the term, or type check it. Recent work does integrate the linear and dependent types (McBride, 2016), by essentially allowing a type-level reference to a linear object but disallowing that reference from being used at run time. By making aliases linear-dependent in this way, we may be able to both restrict cycles and allow typing a dependent heap. Another recent approach integrates dependent types with graded modal types, a variant of linear types in which objects are restricted to a specific number of uses (Orchard and Liepelt, 2017)[1]. This could allow exactly two references, or an unbounded number of type-level references and exactly one run-time reference.

1 Dependent types are listed as future work in that citation, but a prototype language now exists `https://granule-project.github.io/index.html`.

This pass will also be made more complex by recursive functions. We will have to allow some cycles, but only the "well-behaved" ones. This may require an auxiliary judgment, like Coq's guard condition for termination checking, to check that all cycles in the heap are well-defined recursive functions.

CODE GENERATION I conjecture that code generation will be an easy, but tedious, translation. This translation is responsible for making explicit the details such as word sizes, literal values, register values, and which instructions work over which kinds of values. The register file is the only new machine object, but I conjecture that modeling it will be less complex than modeling the heap. The most difficult part will be interpreting assembly code as expressions. We may need to build on recent work on interoperability between functional languages and assembly code (Patterson et al., 2017) However, as the pattern that leads us to this path already forces us to represent individual machine steps in the syntax and typing derivations, it might be simple to translate machine steps into assembly code by this stage.

8.3.2 Practical Considerations

This dissertation has established a theory for type-preserving compilation, but ignored the practice. I have, however, had practical concerns in mind during this dissertation. The following are some considerations ignored earlier in this dissertation, but I believe will be important to make the mythical compiler describing in Chapter 1 a reality.

COMPUTATIONAL RELEVANCE All the translations presented in this dissertation act as if every expression is computationally relevant, but this is an over-approximation. Many of the expressions, terms and types, will be irrelevant at run time. Compiling them into a representation that is effective for a machine to execute is, therefore, a bit silly. For these irrelevant expressions, it would be better to adopt a representation that is small and efficient for type checking, but not necessarily for executing.

There's two good reasons for ignoring computational relevance for now. First, it simplifies designing the prototype dependent-type-preserving compiler. While it's true that some expressions are computationally irrelevant, we still need to figure out how to compile the ones that are relevant. We should do that first, and not prematurely optimize the compiler before we have a fully worked out theory. Second, we don't yet have a good theory for computational relevance in dependent types that we can simply take, off the shelf, and apply to an arbitrary dependently typed (intermediate) language. Research on computational relevance has already produced one Ph.D. dissertation (Mishra-Linger, 2008), without solving the problem for inductive types and dependent pattern matching, and ongoing work is still developing theories for simple core calculi (Tejiŝĉák and Brady, 2015; Nuyts and Devriese, 2018). The state of practice in Coq is to distinguish between base universes, and use a mysterious static analysis at higher universes.

To really address computational relevance, we need a core language with explicit relevance annotations and an elaboration from Coq to the core language. (A complete redesign of computational relevance in existing languages seems unlikely, but I'll keep hoping.) Once we have such a core language with explicit relevance annotations, then we should be able to redesign the type-preserving compilers presented so far (and developed in the future) to selectively compile only relevant expressions. We could also design optimization passes to optimize the irrelevant expressions for type checking.

A selective translation could introduce problems in the interpretation of machine steps, however. For example, if we have two different kinds of functions, irrelevant and relevant, will we ever need to cast from one to another? This seems possible if we decide equivalence by normalization. If not, then it seems likely that we will duplicate code from one level into the other (as already happens in Coq with Set vs Prop), which will exacerbate the code size problem discussed next. However, in some ways, the interpretation of machine steps already casts one kind of computation into another. For example, in CPS, we cast a CPS'd computation into a function application that can be interpreted as an expression. So maybe interpreting irrelevant expressions will not be any more difficult.

However, maybe we do want to compile even irrelevant computations. If we decide equivalence by normalization, we might be able to more efficiently normalize terms by jumping to the assembly code that computes their value, rather than by running an interpreter. This would result in efficient type checking by using normalization by evaluation in assembly. It is unclear whether this is possible, and I don't see how to integrate η-equivalence.

REDUCING CODE SIZE If we are planning to ship compiled code paired with its specification and proofs, code size may become an engineering constraint. My translations have completely ignored this problem, in favor of as many annotations as I need to prove type preservation and decidability. For example, the CPS translation increases annotations by "a lot". Each term becomes annotated with the type from its derivation, which is copied into at least two places: the continuation, and the machine step. This would be unacceptable in practice. So how do we keep specifications and proofs small?

Past work has developed small proof witnesses for dependently typed languages, in particular, Twelf (Sarkar et al., 2005), and we will likely need to adapt this kind of work to the compiler ILs. It's unclear how this work scales to full-spectrum dependently typed languages such as Coq.

One important consideration is whether decreasing the size of proofs will increase the trusted code base, for example by making the proof checker more complex, or increase the type checking time. In many high-assurance scenarios, we might want the proof checker itself to be small. On the other hand, maybe not: we might be able to prove the more complex proof checker correct using a smaller, more easily audited, proof checker. Past work has considered how to balance proof size against proof checker size in the context of PCC and LF (Appel et al., 2003).

Another aspect to this problem is link time efficiency. Given that we want to run type checking at link time to rule out linking errors, we want proof checking to be fast so linking can be fast. Usually there is a space/time trade off, so I would expect that decreasing the size of proofs and specifications will increase the time to check proofs. This could increase link time, which might be undesirable. On the other hand, linking should happen rarely compared to running, so maybe it won't matter.

PRESERVING MEANING OF SPECIFICATIONS One problem I alluded to in Chapter 1 is that we can only stop trusting the compiler if we still trust the specification. This is no problem if we have a specification like $\mathsf{IsCorrect}\ \mathsf{f}$, but what about when the specification is $\boldsymbol{\lambda}\,\mathbf{k}.\,\mathsf{IsCorrect}\ (\boldsymbol{\lambda}\,\mathbf{x}.\,\mathsf{f}^{\div}\ (\boldsymbol{\lambda}\,\mathbf{y}.\,\mathbf{y}\ \mathbf{x}\ \mathbf{k}))$? Compiling specifications, or at least compiling the computations in specifications, makes trusting those specifications much more difficult. One of the fundamental assumptions of type preservation is that *we trust the type translation*. However, when the type translation is also the compiler, because terms are types, doesn't that mean we must either trust the compiler or prove it correct? If so, type preservation alone isn't enough to rule out miscompilation errors, and we must at least prove compiler correctness as well.

This section dedicated to Greg.

We need something somewhat more than type preservation to help preserve the meaning of, or at least our trust in, specifications. At first, I thought it was enough to prove logical consistency of the target. Surely if we were to badly mangle specifications in translation, and prove type preservation, we would inadvertently be inconsistent? I'm no longer sure of that. Greg Morrisett suggested to me that we want some kind of logical implication: reading the types as logical formulas, we want to say the translated type implies the original type. This seem intuitively right, but I'm unsure how to formalize that intuition. Either of these approaches, however, still requires trust in some proof about the compiler, which is unsatisfying to achieving fully *certifying* compilation or proof-carrying code.

A more satisfying approach would be to maintain readability of specifications, so the target specification still reads as $\mathbf{IsCorrect}\ \mathsf{f}^{\div}$. A solution to the computational relevance problem might help by letting us avoid translating irrelevant specifications. However, I'm not sure that this solves the problem in general. Even irrelevant specifications will refer to relevant computations, and it may be difficult to trust that $\mathsf{f}^{\div}$ is the same as the f we expected to be correct, unless we prove compiler correctness. Although I do prove compiler correctness in this dissertation, and it even falls out of the recipe from Chapter 3, I would like to avoid it in favor of a fully certifying compiler using type preservation.

GRADUAL DEPENDENTLY TYPED ASSEMBLY In Chapter 1, I described a mythical compiler able to rule out linking errors when linking with x86 assembly, but all formal definitions of linking in this dissertation rely on type checking, and x86 is not dependently typed. How will we ever interoperate with real assembly? It's not realistic to expect all code to be ported to a dependently typed language and compiled with our compiler, nor to expect all x86 code to be ported to a new dependently typed assembly language (even if it would rule out a lot of errors. Instead, we need some way allow existing code—either x86 assembly or code generated from non-dependently typed languages—to safely interoperate with the dependently typed assembly language generated from our type preserving compiler.

The working conjecture on how to get safe interoperability between a typed and untyped assembly language is *gradual typing* (Ahmed, 2015). The idea is that we will define a gradual type system between typed assembly and assembly. At the boundary between typed and untyped code, the types will become *contracts*—dynamic checks—ensuring safe interoperability. Work on gradual typing suggests this will incur a performance penalty (Takikawa et al., 2016), but static contract verification looks promising and could reduce contract costs to near zero (Tobin-Hochstadt and van Horn, 2012).

To apply gradual typing to a dependent-type preserving compiler, we would need to extend gradual typing to dependent types and to assembly language. There are several instances for gradual typing for forms of dependent types, such as gradual refinement types (Lehmann and Tanter, 2017), and gradual liquid types (Vazou et al., 2018). These seem promising, but probably require nontrivial extensions to support a dependently typed language such as Coq. There is recent work on dynamically typed assembly that should prove a promising starting point for extension to gradual dependently typed assembly (Hernandez, 2018). There is also work on gradual call-by-push-value (CBPV) (New et al., 2019), which could prove a useful starting point since CBPV has been shown to correspond to a low-level SSA IR (Garbuzov et al., 2018).

8.4 Conclusion

In this dissertation, I developed a theory of dependent-type-preservation, showing that type-preservation is a viable way to elimination miscompilation and linking errors from dependently typed languages. I describe the core features of dependency, and essence of type preservation, and a recipe for proving type preservation for dependently typed language. I presented four translations: two I believe will scale to practice as they are, while two will need more work to scale to languages such as Coq. I summarized the lessons of these translations, explain how to apply those lessons to remaining translations required to finish a prototype dependent-type-preserving compiler, and described some practical issues that will need to be solved to develop the mythical full-spectrum type-preserving compiler from Coq to gradually dependently-typed assembly.

LIST OF FIGURES

BIBLIOGRAPHY

D. Adams. *The Restaurant at the End of the Universe*. 1980. ISBN 9780330262132. URL `https://www.worldcat.org/oclc/154399749`.

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.

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP)*, Sept. 2008. doi:10.1145/1411204.1411227.

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

A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundamenta Informaticae*, 77(4), Dec. 2007. doi:10.1007/11417170_22.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6. URL `http://www.worldcat.org/oclc/12285707`.

T. Altenkirch, N. A. Danielsson, A. Löh, and N. Oury. ΠΣ: Dependent types without the sugar. In *International Conference on Functional and Logic Programming*, Apr. 2010. doi:10.1007/978-3-642-12251-4_5.

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, N. Michael, A. Stump, and R. Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31(3–4):231–260, 2003. doi:10.1023/B:JARS.0000021013.61329.58.

B. Barras, P. Corbineau, B. Grégoire, H. Herbelin, and J. L. Sacchini. A new elimination rule for the calculus of inductive constructions. In *International Conference on Types for Proofs and Programs (TYPES)*, Mar. 2008. doi:10.1007/978-3-642-02444-3_3.

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, J. Hatcliff, and M. H. B. Sørensen. CPS translations and applications: The cube and beyond. *Higher-Order and Symbolic Computation*, 12(2), Sept. 1999. doi:10.1023/a:1010000206149.

G. Barthe, J. Hatcliff, and M. H. Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1-2), Oct. 2001. doi:10.1016/s0304-3975(01)00012-3.

J. Bernardy, P. Jansson, and R. Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming (JFP)*, 22(02), Mar. 2012. doi:10.1017/S0956796812000056.

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 2018. doi:10.1145/3192366.3192372.

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.

L. Cardelli. A polymorphic λ-calculus with type:type. Technical Report 10, Digital Equipment Corporation. Systems Research Center., 1986. URL `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-10.pdf`.

C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2014. doi:10.1145/2535838.2535883.

J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *International Conference on Programming Language Design and Implementation (PLDI)*, June 2010. doi:10.1145/1806596.1806643.

J. Cockx, D. Devriese, and F. Piessens. Unifiers as equivalences: Proof-relevant unification of dependently typed data. In *International Conference on Functional Programming (ICFP)*, Sept. 2016. doi:10.1145/2951913.2951917.

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.

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. An analysis of Girard's paradox. In *Symposium on Logic in Computer Science (LICS)*, July 1986. URL `https://hal.inria.fr/inria-00076023`.

T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, Sept. 1989. URL `https://hal.inria.fr/inria-00075471`.

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.

P. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming (ICFP)*, Sept. 2000. doi:10.1145/357766.351262.

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.

D. Garbuzov, W. Mansky, C. Rizkallah, and S. Zdancewic. Structural operational semantics for control flow graph machines. *CoRR*, abs/1805.05400, 2018. URL `http://arxiv.org/abs/1805.05400`.

J. H. Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993. URL `http://www.ru.nl/publish/pages/682191/geuvers_jh.pdf`.

J. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. URL `https://www.worldcat.org/oclc/493768392`.

J. Girard. A new constructive logic: classical logic. *Mathematical Structures in Computer Science*, 1(3), Nov. 1991. doi:10.1017/S0960129500001328.

J. Gross, A. Chlipala, and D. I. Spivak. Experience implementing a performant category-theory library in Coq. In *International Conference on Interactive Theorem Proving (ITP)*, July 2014. doi:10.1007/978-3-319-08970-6_18.

R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 1994. doi:10.1145/174675.176927.

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.

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

I. Hernandez. Strong-dism: A first attempt to a dynamically typed assembly language (d-tal). Master's thesis, University of South Florida, 2018. URL https://scholarcommons.usf.edu/etd/7033.

L. Jia, J. Zhao, V. Sjöberg, and S. Weirich. Dependent types and program equivalence. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2010. doi:10.1145/1706299.1706333.

J. Kang, Y. Kim, C. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2016. doi:10.1145/2837614.2837642.

C. Keller and M. Lasson. Parametricity in an impredicative sort. In *International Workshop on Computer Science Logic (CSL)*, Sept. 2012. URL https://hal.inria.fr/hal-00730913.

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

A. Kovács. Closure conversion for dependent type theory with type-passing polymorphism. In *International Workshop on Types for Proofs and Programs (TYPES)*, 2018. URL https://github.com/AndrasKovacs/misc-stuff/blob/master/MemControl/types2018/abstract-types-2018-cconv.pdf.

R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2017. doi:10.1145/3093333.3009855.

N. R. Krishnaswami and D. Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *International Workshop on Computer Science Logic (CSL)*, Sept. 2013. doi:10.4230/LIPIcs.CSL.2013.432.

R. Kumar, M. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2014. doi:10.1145/2535838.2535841.

N. Lehmann and E. Tanter. Gradual refinement types. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2017. doi:10.1145/3093333.3009856.

X. Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 1992. doi:10.1145/143165.143205.

X. Leroy. An overview of types in compilation. In *Workshop on Types in Compilation (TIC)*, volume 1743, pages 1–8, Mar. 1998. doi:10.1007/BFb0055509.

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

M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, May 1997. URL http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-122.pdf.

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/.

P. Martin-Löf. A theory of types. Revised Oct. 1971. Privately circulated manuscript, Feb. 1971.

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.

C. McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Workshop on Generic Programming (WGP)*, Sept. 2010. doi:10.1145/1863495.1863497.

C. McBride. I got plenty o' nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Mar. 2016. doi:10.1007/978-3-319-30936-1_12.

Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 1996. doi:10.1145/237721.237791.

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.

É. Miquey. A sequent calculus with dependent types for classical arithmetic. In *Symposium on Logic in Computer Science (LICS)*, July 2018. doi:10.1145/3209108.3209199.

R. N. Mishra-Linger. *Irrelevance, Polymorphism, and Erasure in Type Theory*. PhD thesis, Portland State University, Nov. 2008. doi:10.15760/etd.2669.

G. Morrisett and R. Harper. Typed closure conversion for recursively-defined functions. *Electronic Notes in Theoretical Computer Science*, 10, June 1998. doi:10.1016/s1571-0661(05)80702-9.

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.

A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard Univeristy, Dec. 2005. URL http://nrs.harvard.edu/urn-3:HUL.InstRepos:26506436. Update Jan. 2006.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *International Conference on Functional Programming (ICFP)*, Sept. 2006. doi:10.1145/1159803.1159812.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming (ICFP)*, Sept. 2008. doi:10.1145/1411204.1411237.

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 P. Lee. The design and implementation of a certifying compiler. In *International Conference on Programming Language Design and Implementation (PLDI)*, May 1998. doi:10.1145/277652.277752.

G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *International Conference on Functional Programming (ICFP)*, Sept. 2015. doi:10.1145/2784731.2784764.

M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming (ICFP)*, Sept. 2016. doi:10.1145/2951913.2951941.

M. S. New, D. R. Licata, and A. Ahmed. Gradual type theory. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3(POPL), Jan. 2019. doi:10.1145/3290328.

A. Nuyts and D. Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In *Symposium on Logic in Computer Science (LICS)*, July 2018. doi:10.1145/3209108.3209119.

A. Nuyts, A. Vezzosi, and D. Devriese. Parametric quantifiers for dependent type theory. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP), Aug. 2017. doi:10.1145/3110276.

D. Orchard and V. Liepelt. Gram: A linear functional language with graded modal types. In *Trends in Linear Logic and Applications (TLLA)*, Sept. 2017. URL http://www.cs.ox.ac.uk/conferences/fscd2017/preproceedings_unprotected/TLLA_Orchard.pdf.

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

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. Pédrot and N. Tabareau. An effectful way to eliminate addiction to dependence. In *Symposium on Logic in Computer Science (LICS)*, Jan. 2017. doi:10.1109/lics.2017.8005113.

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

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

G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2), Dec. 1975. doi:10.1016/0304-3975(75)90017-1.

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.

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

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.

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

Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1), Jan. 2005. doi:10.1145/1053468.1053469.

M. Sozeau and N. Tabareau. Universe polymorphism in Coq. In *International Conference on Interactive Theorem Proving (ITP)*, July 2014. doi:10.1007/978-3-319-08970-6_32.

G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Symposium on Principles of Programming Languages (POPL)*, Jan. 2015. doi:10.1145/2676726.2676985.

T. Streicher. Independence results for calculi of dependent types. In *Category Theory and Computer Science*, Sept. 1989. doi:10.1007/BFb0018350.

N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *Journal of Functional Programming (JFP)*, 23(4), July 2013. doi:10.1017/S0956796813000142.

A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Symposium on Principles of Programming Languages (POPL)*, 2016. doi:10.1145/2837614.2837630.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *International Conference on Programming Language Design and Implementation (PLDI)*, May 1996. doi:10.1145/231379.231414.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed, optimizing compiler for ML. *ACM SIGPLAN Notices, 20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*, 39(4), Apr. 2004. doi:10.1145/989393.989449.

M. Tejiŝĉák and E. Brady. Practical erasure in dependently typed languages. Accessed Nov. 2018, Feb. 2015. URL `https://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf`.

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

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

K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), Aug. 1984. doi:10.1145/358198.358210.

A. Timany and B. Jacobs. Category theory in Coq 8.5. *CoRR*, May 2015. URL `https://arxiv.org/abs/1505.06430`.

S. Tobin-Hochstadt and D. van Horn. Higher-order symbolic execution via contracts. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2012. doi:10.1145/2384616.2384655.

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

N. Vazou, É. Tanter, and D. van Horn. Gradual liquid type inference. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(OOPSLA), Nov. 2018. doi:10.1145/3276502.

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.

S. Weirich, A. Voizard, P. H. A. de Amorim, and R. A. Eisenberg. A specification for dependent types in Haskell. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP), Aug. 2017. doi:10.1145/3110275.

H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. URL `https://www.cs.bu.edu/~hwxi/academic/papers/DML-thesis.pdf`.

H. Xi and R. Harper. A dependently typed assembly language. In *International Conference on Functional Programming (ICFP)*, Sept. 2001. doi:10.1145/507635.507657.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*, June 2011. doi:10.1145/1993498.1993532.

A REFERENCE FOR ECC^D

This appendix contains the complete definitions of the semantics of ECC^D, in particular, its type system and evaluation semantics. All ECC^D figures from Chapter 2 are reproduced and completed with elided parts here, and elided figures are presented here.

Note that the observations depend on the particular pair languages and translation; Here, I give only ECC^D observations. In each target language, I will define the target language observation, and each translation gives the cross-language relation for observations.

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

Figure A.1: ECC^D Syntax

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

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

Figure A.2: ECC^D Reduction

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x = e' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \rhd^* \text{let } x = e_1' \text{ in } e_2'}\ \textsc{Red-Cong-Let}$$

Figure A.3: ECC^D Conversion

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

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

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

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

Figure A.4: ECC^D Equivalence

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

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

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

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

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

Figure A.5: ECC^D Subtyping

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

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

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

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

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

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

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False}$$

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

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

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

$\boxed{\vdash \Gamma}$

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

Figure A.6: ECC^D Typing

$$\textit{ECC}^D\ \textit{Observations} \quad v \quad ::= \quad \mathsf{true} \mid \mathsf{false}$$

Figure A.7: ECC^D Observations

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

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool}}{\Gamma \vdash \mathsf{e}}$$

$\boxed{\vdash \mathsf{e}}$

$$\frac{\cdot \vdash \mathsf{e}}{\vdash \mathsf{e}}$$

Figure A.8: ECC^D Components and Programs

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

$$\mathsf{eval}(\mathsf{e}) \quad = \quad \mathsf{v} \quad \text{if} \vdash \mathsf{e} \text{ and } \cdot \vdash \mathsf{e} \triangleright^* \mathsf{v}$$

Figure A.9: ECC^D Evaluation

$$\textit{Closing Substitutions} \quad \gamma \quad \overset{\text{def}}{=} \quad \cdot \mid \gamma[\mathsf{x} \mapsto \mathsf{e}]$$

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

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

$\boxed{\gamma(\mathsf{e}) = \mathsf{e}}$

$$\cdot(\mathsf{e}) = \mathsf{e} \qquad \gamma[\mathsf{x} \mapsto \mathsf{e}'](\mathsf{e}) = \gamma(\mathsf{e}[\mathsf{x}/\mathsf{e}'])$$

Figure A.10: ECC^D Closing Substitutions and Linking

B REFERENCE FOR ECC^A

This appendix contains the complete definitions for ECC^A. All ECC^A figures from Chapter 4 are reproduced and completed with elided parts here, and elided figures are presented here.

Recall that ECC^A is a syntactic restriction of ECC^D that supports a machine evaluation semantics. The primary difference between the figure here and in Chapter 4 is the syntax is extended with non-ANF expressions to formally describe type checking in ECC^A. Recall that the type system for ECC^A is the same as ECC^D, so the figures related to the type system are identical to those in Appendix A.

Typographical Note. *In this appendix, I typeset ECC^A in a* **bold, red, serif font***.*

$$
\begin{array}{llcl}
\textit{Universes} & \mathbf{U} & ::= & \mathbf{Prop} \mid \mathbf{Type}_i \\
\textit{Values} & \mathbf{V} & ::= & \mathbf{x} \mid \mathbf{U} \mid \mathbf{\Pi\, x : M.\, M} \mid \mathbf{\lambda\, x : M.\, M} \mid \mathbf{\Sigma\, x : M.\, M} \\
 & & \mid & \mathbf{\langle V, V\rangle\, as\, M} \\
\textit{Computations} & \mathbf{N} & ::= & \mathbf{V} \mid \mathbf{V\ V} \mid \mathbf{fst\ V} \mid \mathbf{snd\ V} \\
\textit{Configurations} & \mathbf{M} & ::= & \mathbf{N} \mid \mathbf{let\, x = N\, in\, M} \\
\textit{Continuations} & \mathbf{K} & ::= & [\cdot] \mid \mathbf{let\, x =} [\cdot] \mathbf{\, in\, M} \\
\textit{Expressions} & \mathbf{e, A, B} & ::= & \mathbf{x} \mid \mathbf{U} \mid \mathbf{\Pi\, x : A.\, B} \mid \mathbf{\lambda\, x : A.\, e} \mid \mathbf{e\ e} \mid \mathbf{\Sigma\, x : A.\, B} \\
 & & \mid & \mathbf{\langle e_1, e_2\rangle\, as\, \Sigma\, x : A.\, B} \mid \mathbf{fst\, e} \mid \mathbf{snd\, e} \mid \mathbf{bool} \mid \mathbf{true} \\
 & & \mid & \mathbf{false} \mid \mathbf{if\, e\, then\, e_1\, else\, e_2} \mid \mathbf{let\, x = e\, in\, e} \\
\textit{Environments} & \mathbf{\Gamma} & ::= & \cdot \mid \mathbf{\Gamma, x : A} \mid \mathbf{\Gamma, x = e}
\end{array}
$$

Figure B.1: ECC^A Syntax

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

$$
\begin{array}{rll}
\Gamma \vdash (\lambda\, x : A.\, e_1)\ e_2 & \triangleright_\beta & e_1[e_2/x] \\
\Gamma \vdash \mathsf{fst}\ \langle e_1, e_2 \rangle & \triangleright_{\pi_1} & e_1 \\
\Gamma \vdash \mathsf{snd}\ \langle e_1, e_2 \rangle & \triangleright_{\pi_2} & e_2 \\
\Gamma \vdash \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \triangleright_{\iota_1} & e_1 \\
\Gamma \vdash \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \triangleright_{\iota_1} & e_2 \\
\Gamma \vdash x & \triangleright_\delta & e \qquad \text{where } x = e \in \Gamma \\
\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' & \triangleright_\zeta & e[e'/x]
\end{array}
$$

Figure B.2: ECC^A Reduction

$$\frac{}{\Gamma \vdash e \triangleright^* e}\ \textsc{Red-Refl} \qquad \frac{\Gamma \vdash e \triangleright e_1 \qquad \Gamma \vdash e_1 \triangleright^* e'}{\Gamma \vdash e \triangleright^* e'}\ \textsc{Red-Trans}$$

$$\frac{\Gamma \vdash A \triangleright^* A' \qquad \Gamma, x : A' \vdash e \triangleright^* e'}{\Gamma \vdash \Pi\, x : A.\, e \triangleright^* \Pi\, x : A'.\, e'}\ \textsc{Red-Cong-Pi}$$

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

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

$$\frac{\Gamma \vdash A \triangleright^* A' \qquad \Gamma, x : A' \vdash e \triangleright^* e'}{\Gamma \vdash \Sigma\, x : A.\, e \triangleright^* \Sigma\, x : A'.\, e'}\ \textsc{Red-Cong-Sig}$$

$$\frac{\Gamma \vdash e_1 \triangleright^* e_1' \qquad \Gamma \vdash e_2 \triangleright^* e_2' \qquad \Gamma \vdash A \triangleright^* A'}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathsf{as}\ A \triangleright^* \langle e_1', e_2' \rangle\ \mathsf{as}\ A'}\ \textsc{Red-Cong-Pair}$$

$$\frac{\Gamma \vdash e \triangleright^* e'}{\Gamma \vdash \mathsf{fst}\ e \triangleright^* \mathsf{fst}\ e'}\ \textsc{Red-Cong-Fst} \qquad \frac{\Gamma \vdash e \triangleright^* e'}{\Gamma \vdash \mathsf{snd}\ e \triangleright^* \mathsf{snd}\ e'}\ \textsc{Red-Cong-Snd}$$

$$\frac{\Gamma \vdash e \triangleright^* e' \qquad \Gamma \vdash e_1 \triangleright^* e_1' \qquad \Gamma \vdash e_2 \triangleright^* e_2'}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \triangleright^* \mathsf{if}\ e'\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2'}\ \textsc{Red-Cong-If}$$

$$\frac{\Gamma \vdash e_1 \triangleright^* e_1' \qquad \Gamma, x = e' \vdash e_2 \triangleright^* e_2'}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleright^* \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}\ \textsc{Red-Cong-Let}$$

Figure B.3: ECC^A Conversion

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

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

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

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

Figure B.4: ECC^A Equivalence

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

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

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

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

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

Figure B.5: ECC^A Subtyping

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Prop} : \mathbf{Type}_0}\ \textsc{Prop} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Type}_i : \mathbf{Type}_{i+1}}\ \textsc{Type}$$

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

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_i}{\Gamma \vdash \Pi\, x : A.\, B : \mathbf{Type}_i}\ \textsc{Pi-Type}$$

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

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_i}{\Gamma \vdash \Sigma\, x : A.\, B : \mathbf{Type}_i}\ \textsc{Sig}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathbf{as}\ (\Sigma\, x : A.\, B) : \Sigma\, x : A.\, B}\ \textsc{Pair} \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathbf{fst}\ e : A}\ \textsc{Fst}$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathbf{snd}\ e : B[\mathbf{fst}\ e/x]}\ \textsc{Snd} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{bool} : \mathbf{Prop}}\ \textsc{Bool} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}\ \textsc{True}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}\ \textsc{False}$$

$$\frac{\begin{array}{c}\Gamma, y : \mathbf{bool} \vdash B : U \\ \Gamma \vdash e : \mathbf{bool} \qquad \Gamma \vdash e_1 : B[\mathbf{true}/y] \qquad \Gamma \vdash e_2 : B[\mathbf{false}/y]\end{array}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : B[e/y]}\ \textsc{If}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A, x = e \vdash e' : B}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : B[e/x]}\ \textsc{Let}$$

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

$\boxed{\vdash \Gamma}$

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

Figure B.6: ECC^{A} Typing

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

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

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

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

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

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

Figure B.7: ECC^{A} Composition of Configurations

$\boxed{\mathbf{\Gamma} \vdash \mathbf{K} : (\mathbf{N} : \mathbf{M}_A) \Rightarrow \mathbf{M}_B}$

$$\frac{}{\mathbf{\Gamma} \vdash [\cdot] : (\mathbf{N} : \mathbf{M}_A) \Rightarrow \mathbf{M}_A}\ \text{K-EMPTY}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{N} : \mathbf{M}_A \qquad \mathbf{\Gamma}, \mathbf{y} = \mathbf{N} \vdash \mathbf{M} : \mathbf{M}_B}{\mathbf{\Gamma} \vdash \mathbf{let}\,\mathbf{y} = [\cdot]\,\mathbf{in}\,\mathbf{M} : (\mathbf{N} : \mathbf{M}_A) \Rightarrow \mathbf{M}_B}\ \text{K-BIND}$$

Figure B.8: ECC^{A} Continuation Typing

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

$$\texttt{defs}(\mathbf{M}) = \mathbf{x}_1 = \mathbf{N}_1, \dots, \mathbf{x}_n = \mathbf{N}_n \quad \text{where}\,\mathbf{M} = \mathbf{let}\,\mathbf{x}_1 {=} \mathbf{N}_1\,\mathbf{in}\ \dots \mathbf{let}\,\mathbf{x}_n {=} \mathbf{N}_n\,\mathbf{in}\,\mathbf{N}_{n+1}$$

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

$$\texttt{hole}(\mathbf{M}) = \mathbf{N}_{n+1} \qquad \text{where } \mathbf{M} = \mathbf{let}\,\mathbf{x}_1 = \mathbf{N}_1\,\mathbf{in}\ \dots \mathbf{let}\,\mathbf{x}_n = \mathbf{N}_n\,\mathbf{in}\,\mathbf{N}_{n+1}$$

Figure B.9: ECC^{A} Continuation Exports

$\boxed{\mathbf{\Gamma} \vdash \mathbf{M}}$ $\boxed{\vdash \mathbf{M}}$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{M} : \mathbf{bool}}{\mathbf{\Gamma} \vdash \mathbf{M}} \qquad\qquad \frac{\cdot \vdash \mathbf{M}}{\vdash \mathbf{M}}$$

Figure B.10: ECC^{A} Components and Programs

$$\textit{ECC}^A\ \textit{Observations} \quad \mathbf{v} \quad ::= \quad \mathbf{true} \mid \mathbf{false}$$

Figure B.11: ECC^A Observations

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

$$\begin{array}{rcl}
\mathbf{K}[(\lambda\, \mathbf{x} : \mathbf{A}.\, \mathbf{M})\ \mathbf{V}] & \mapsto_\beta & \mathbf{K}\langle\!\langle \mathbf{M}[\mathbf{V}/\mathbf{x}]\rangle\!\rangle \\
\mathbf{K}[\mathbf{fst}\ \langle \mathbf{V}_1, \mathbf{V}_2\rangle] & \mapsto_{\pi_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{snd}\ \langle \mathbf{V}_1, \mathbf{V}_2\rangle] & \mapsto_{\pi_2} & \mathbf{K}[\mathbf{V}_2] \\
\mathbf{let}\ \mathbf{x} = \mathbf{V}\ \mathbf{in}\ \mathbf{M} & \mapsto_\zeta & \mathbf{M}[\mathbf{V}/\mathbf{x}]
\end{array}$$

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

$$\frac{}{\vdash \mathbf{M} \mapsto^* \mathbf{M}}\ \textsc{RedA-Refl} \qquad \frac{\mathbf{M} \mapsto \mathbf{M}_1 \qquad \vdash \mathbf{M}_1 \mapsto^* \mathbf{M}'}{\vdash \mathbf{M} \mapsto^* \mathbf{M}'}\ \textsc{RedA-Trans}$$

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

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

Figure B.12: ECC^A Evaluation

$$\textit{Closing Substitutions} \quad \gamma \quad \overset{\text{def}}{=} \quad \cdot \mid \gamma[\mathbf{x} \mapsto \mathbf{M}]$$

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

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

$\boxed{\gamma(\mathbf{M}) = \mathbf{M}}$

$$\cdot(\mathbf{M}) = \mathbf{M} \qquad \gamma[\mathbf{x} \mapsto \mathbf{M}'](\mathbf{M}) = \gamma(\mathbf{M}[\mathbf{x}/\!/\mathbf{M}'])$$

Figure B.13: ECC^A Closing Substitutions and Linking

C REFERENCE FOR ANF TRANSLATION

This appendix contains the complete definitions for the ANF translation from Chapter 4. All figures related to the ANF translation are reproduced and completed with elided parts here, and elided figures are presented here.

Typographical Note. *In this appendix, I typeset the source language, ECC^D, in a* blue, non-bold, sans-serif font, *and the target language, ECC^A, in a* **bold, red, serif font**.

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

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

Figure C.1: Observation Relation between ECC^D and ECC^A

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

$$\begin{array}{rcl}
[\![\mathsf{e}]\!] & \overset{\text{def}}{=} & [\![\mathsf{e}]\!]\ [\cdot] \\
[\![\mathsf{x}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & \mathbf{K}[\mathbf{x}] \\
[\![\mathsf{Prop}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & \mathbf{K}[\mathbf{Prop}] \\
[\![\mathsf{Type}_i]\!]\ \mathbf{K} & \overset{\text{def}}{=} & \mathbf{K}[\mathbf{Type}_i] \\
[\![\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & \mathbf{K}[\boldsymbol{\Pi}\,\mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!]] \\
[\![\lambda\,\mathsf{x} : \mathsf{A}.\,\mathsf{e}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & \mathbf{K}[\boldsymbol{\lambda}\,\mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{e}]\!]] \\
[\![\mathsf{e}_1\ \mathsf{e}_2]\!]\ \mathbf{K} & \overset{\text{def}}{=} & [\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ ([\![\mathsf{e}_2]\!]\ \mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{x}_1\ \mathbf{x}_2]) \\
[\![\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & \mathbf{K}[\boldsymbol{\Sigma}\,\mathbf{x} : [\![\mathsf{A}]\!].\,[\![\mathsf{B}]\!]] \\
[\![\langle \mathsf{e}_1, \mathsf{e}_2 \rangle\ \mathsf{as}\ \mathsf{A}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & [\![\mathsf{e}_1]\!]\ \mathbf{let}\ \mathbf{x}_1 = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}_2]\!]\ (\mathbf{let}\ \mathbf{x}_2 = [\cdot]\ \mathbf{in}\ \mathbf{K}[(\langle \mathbf{x}_1, \mathbf{x}_2 \rangle\ \mathbf{as}\ [\![\mathsf{A}]\!])]) \\
[\![\mathsf{fst}\ \mathsf{e}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{fst}\ \mathbf{x}] \\
[\![\mathsf{snd}\ \mathsf{e}]\!]\ \mathbf{K} & \overset{\text{def}}{=} & [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ \mathbf{K}[\mathbf{snd}\ \mathbf{x}] \\
[\![\mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}']\!]\ \mathbf{K} & \overset{\text{def}}{=} & [\![\mathsf{e}]\!]\ \mathbf{let}\ \mathbf{x} = [\cdot]\ \mathbf{in}\ [\![\mathsf{e}']\!]\ \mathbf{K}
\end{array}$$

Figure C.2: ANF Translation from ECC^D to ECC^A

D | REFERENCE FOR ECC^{CC}

This appendix contains the complete definitions for ECC^{CC}. All ECC^{CC} figures from Chapter 5 are reproduced and completed with elided parts here, and elided figures are presented here. In particular, additional figures regarding the ANF variant of ECC^{CC} are included here.

Typographical Note. *In this appendix, I typeset ECC^{CC} in a* **bold, red, serif font**, *and typeset the model language ECC^{D} in a* blue, non-bold, sans-serif font.

$$\begin{array}{llcl}
\textit{Universes} & \mathbf{U} & ::= & \mathbf{Prop} \mid \mathbf{Type}_i \\
\textit{Expressions} & \mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \mathbf{U} \mid \mathbf{Code}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{B} \mid \boldsymbol{\lambda}\,(\mathbf{x}' : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{e} \\
 & & \mid & \boldsymbol{\Pi}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \mid \langle\!\langle \mathbf{e}, \mathbf{e}' \rangle\!\rangle \mid \mathbf{e}\;\mathbf{e}' \mid \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \\
 & & \mid & \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\;\mathbf{as}\;\boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \mid \mathbf{fst}\;\mathbf{e} \mid \mathbf{snd}\;\mathbf{e} \mid \mathbf{bool} \mid \mathbf{true} \\
 & & \mid & \mathbf{false} \mid \mathbf{if}\;\mathbf{e}\;\mathbf{then}\;\mathbf{e}_1\;\mathbf{else}\;\mathbf{e}_2 \mid \mathbf{let}\;\mathbf{x} = \mathbf{e}\;\mathbf{in}\;\mathbf{e} \mid \mathbf{1} \mid \langle\rangle
\end{array}$$

Figure D.1: ECC^{CC} Syntax

$$\begin{array}{rcl}
\langle \mathbf{e}_1, \mathbf{e}_2 \rangle & = & \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\;\mathbf{as}\;\boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \\
 & & \text{where } \mathbf{e}_1 : \mathbf{A} \text{ and } \mathbf{e}_2 : \mathbf{B} \text{ are inferred from context} \\
\langle \mathbf{e}_i ... \rangle & = & \langle \mathbf{e}_0, \langle \mathbf{e}_1, ... \langle \mathbf{e}_n, \langle\rangle \rangle\rangle\rangle \\
\boldsymbol{\Sigma}\,(\mathbf{x}_i : \mathbf{A}_i ...) & = & \boldsymbol{\Sigma}\,\mathbf{x}_0 : \mathbf{A}_0.\,\boldsymbol{\Sigma}\,\mathbf{x}_1 : \mathbf{A}_1. ...\boldsymbol{\Sigma}\,\mathbf{x}_n : \mathbf{A}_n.\,\mathbf{1} \\
\mathbf{let}\;\langle \mathbf{x}_i ... \rangle = \mathbf{e}\;\mathbf{in}\;\mathbf{e}' & = & \mathbf{let}\;\mathbf{x}_0 = \mathbf{fst}\;\mathbf{e}\;\mathbf{in} \\
 & & \quad \mathbf{let}\;\mathbf{x}_1 = \mathbf{fst}\;\mathbf{snd}\;\mathbf{e}\;\mathbf{in}\;... \\
 & & \qquad \mathbf{let}\;\mathbf{x}_n = \mathbf{fst}\;\mathbf{snd}\;...\mathbf{snd}\;\mathbf{e}\;\mathbf{in}\;\mathbf{e}'
\end{array}$$

Figure D.2: ECC^{CC} Syntactic Sugar

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

$$\begin{array}{rcll}
\Gamma \vdash \langle\!\langle \lambda\, x' : A', x : A.\, e_1, e' \rangle\!\rangle\; e & \rhd_\beta & e_1[e'/x'][e/x] & \\
\Gamma \vdash \mathsf{fst}\ \langle e_1, e_2 \rangle & \rhd_{\pi_1} & e_1 & \\
\Gamma \vdash \mathsf{snd}\ \langle e_1, e_2 \rangle & \rhd_{\pi_2} & e_2 & \\
\Gamma \vdash \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_1} & e_1 & \\
\Gamma \vdash \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \rhd_{\iota_2} & e_2 & \\
\Gamma \vdash x & \rhd_\delta & e & \text{where } x = e \in \Gamma \\
\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e_1 & \rhd_\zeta & e_1[e/x] &
\end{array}$$

Figure D.3: ECC^{CC} Reduction

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

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \rhd^* A_1' \\ \Gamma, n : A_1' \vdash A_2 \rhd^* A_2' \qquad \Gamma, n : A_1', x : A_2' \vdash B \rhd^* B'\end{array}}{\Gamma \vdash \mathbf{Code}\,(n : A_1, x : A_2).\,B \rhd^* \mathbf{Code}\,(n : A_1', x : A_2').\,B'}\ \textsc{Red-Cong-T-Code}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \rhd^* A_1' \\ \Gamma, n : A_1' \vdash A_2 \rhd^* A_2' \qquad \Gamma, n : A_1', x : A_2' \vdash e \rhd^* e'\end{array}}{\Gamma \vdash \lambda\,(n : A_1, x : A_2).\,e \rhd^* \lambda\,(n : A_1', x : A_2').\,e'}\ \textsc{Red-Cong-Code}$$

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \langle\!\langle e_1, e_2 \rangle\!\rangle \rhd^* \langle\!\langle e_1', e_2' \rangle\!\rangle}\ \textsc{Red-Cong-Clo}$$

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

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

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

$$\frac{\Gamma \vdash e \rhd^* e'}{\Gamma \vdash \mathbf{fst}\, e \rhd^* \mathbf{fst}\, e'}\ \textsc{Red-Cong-Fst} \qquad \frac{\Gamma \vdash e \rhd^* e'}{\Gamma \vdash \mathbf{snd}\, e \rhd^* \mathbf{snd}\, e'}\ \textsc{Red-Cong-Snd}$$

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x = e' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \rhd^* \mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2'}\ \textsc{Red-Cong-Let}$$

Figure D.4: ECC^{CC} Conversion

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rhd^* \langle\langle \lambda\,(x' : A', x : A).\, e_1', e' \rangle\rangle \\ \Gamma \vdash e_2 \rhd^* e_2' \qquad \Gamma, x : A \vdash e_1[e'/x'] \equiv e_2'\ x\end{array}}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-}\textsc{Clo}_1$$

$$\frac{\begin{array}{c}\Gamma \vdash e_2 \rhd^* \langle\langle \lambda\,(x' : A', x : A).\, e_2', e' \rangle\rangle \\ \Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x : A \vdash e_1'\ x \equiv e_2'[e'/x']\end{array}}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-}\textsc{Clo}_2$$

Figure D.5: ECC^{CC} Equivalence

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \equiv A_2 \\ \Gamma, n_1 : A_1 \vdash A_1' \equiv A_2' \qquad \Gamma, n_1 : A_1, x_1 : A_1' \vdash B_1 \preceq B_2[n_1/n_2][x_1/x_2]\end{array}}{\Gamma \vdash \mathbf{Code}\,(n_1 : A_1, x_1 : A_1').\, B_1 \preceq \mathbf{Code}\,(n_2 : A_2, x_2 : A_2').\, B_2}\ \preceq\text{-}\textsc{Code}$$

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

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

Figure D.6: ECC^{CC} Subtyping

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

$$\frac{x : A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Prop} : \mathbf{Type}_1}\ \textsc{Prop}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Type}_i : \mathbf{Type}_{i+1}}\ \textsc{Type} \qquad \frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : \mathbf{Prop}}{\Gamma \vdash \Pi\, x : A.\, B : \mathbf{Prop}}\ \textsc{Pi-Prop}$$

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_i}{\Gamma \vdash \Pi\, x : A.\, B : \mathbf{Type}_i}\ \textsc{Pi-Type}$$

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

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_i}{\Gamma \vdash \Sigma\, x : A.\, B : \mathbf{Type}_i}\ \textsc{Sig}$$

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

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathbf{snd}\ e : B[\mathbf{fst}\ e/x]}\ \textsc{Snd} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{bool} : \mathbf{Prop}}\ \textsc{Bool} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}\ \textsc{True}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}\ \textsc{False}$$

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

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A, x = e \vdash e' : B}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : B[e/x]}\ \textsc{Let}$$

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

Figure D.7: ECC^{CC} Typing (1/2)

$$\frac{\Gamma, x' : A', x : A \vdash B : \mathbf{Prop}}{\Gamma \vdash \mathbf{Code}\,(x' : A', x : A).\, B : \mathbf{Prop}}\ \textsc{T-Code-Prop}$$

$$\frac{\Gamma, x' : A', x : A \vdash B : \mathbf{Type}_i}{\Gamma \vdash \mathbf{Code}\,(x : A, x' : A').\, B : \mathbf{Type}_i}\ \textsc{T-Code-Type}$$

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

$$\frac{\Gamma \vdash e : \mathbf{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash \langle\!\langle e, e' \rangle\!\rangle : \Pi\, x : A[e'/x'].\, B[e'/x']}\ \textsc{Clo} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{1} : \mathbf{Prop}}\ \textsc{T-Unit}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textsc{Unit}$$

Figure D.8: ECC^{CC} Typing (2/2)

$\boxed{\vdash \Gamma}$

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

Figure D.9: ECC^{CC} Well-formed Environments

$$\textit{ECC}^{CC}\ \textit{Observations} \quad v \quad ::= \quad \mathbf{true} \mid \mathbf{false}$$

Figure D.10: ECC^{CC} Observations

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

$$\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash e} \qquad \frac{\cdot \vdash e}{\vdash e}$$

Figure D.11: ECC^{CC} Components and Programs

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

$$\mathbf{eval}(e) \quad = \quad v \quad \text{if} \vdash e \text{ and } e \triangleright^* v$$

Figure D.12: ECC^{CC} Evaluation

$$\textit{Closing Substitutions} \quad \gamma \overset{\text{def}}{=} \cdot \mid \gamma[x \mapsto e]$$

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

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

$$\boxed{\gamma(e) = e}$$

$$\cdot(e) = e \qquad \gamma[x \mapsto e'](e) = \gamma(e[x/e'])$$

Figure D.13: ECC^{CC} Closing Substitutions and Linking

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

$$\begin{aligned}
[\![\mathbf{Prop}]\!]^\circ &\overset{\text{def}}{=} \mathsf{Prop} \\
[\![\mathbf{Type}_i]\!]^\circ &\overset{\text{def}}{=} \mathsf{Type}_i \\
[\![x]\!]^\circ &\overset{\text{def}}{=} x \\
[\![1]\!]^\circ &\overset{\text{def}}{=} \Pi\, \alpha : \mathsf{Prop}\,.\, \Pi\, x : \alpha.\, \alpha \\
[\![\langle\rangle]\!]^\circ &\overset{\text{def}}{=} \lambda\, \alpha : \mathsf{Prop}\,.\, \lambda\, x : \alpha.\, x \\
[\![\mathbf{let}\, x = e\, \mathbf{in}\, e']\!]^\circ &\overset{\text{def}}{=} \mathsf{let}\, x = [\![e]\!]^\circ\ \mathsf{in}\ [\![e']\!]^\circ \\
[\![\Pi\, x : A.\, B]\!]^\circ &\overset{\text{def}}{=} \Pi\, x : [\![A]\!]^\circ.\, [\![B]\!]^\circ \\
[\![\langle\!\langle e, e' \rangle\!\rangle]\!]^\circ &\overset{\text{def}}{=} [\![e]\!]^\circ\ \ [\![e']\!]^\circ \\
[\![\mathbf{Code}\,(x' : A', x : A).\, B]\!]^\circ &\overset{\text{def}}{=} \Pi\, x' : [\![A']\!]^\circ.\, \Pi\, x : [\![A]\!]^\circ.\, [\![B]\!]^\circ \\
[\![\boldsymbol{\lambda}\,(x' : A', x : A).\, e]\!]^\circ &\overset{\text{def}}{=} \lambda\, x' : [\![A']\!]^\circ.\, \lambda\, x : [\![A]\!]^\circ.\, [\![e]\!]^\circ \\
[\![e\ e']\!]^\circ &\overset{\text{def}}{=} [\![e]\!]^\circ\ \ [\![e']\!]^\circ \\
[\![\mathbf{bool}]\!]^\circ &\overset{\text{def}}{=} \mathsf{bool} \\
[\![\mathbf{true}]\!]^\circ &\overset{\text{def}}{=} \mathsf{true} \\
[\![\mathbf{false}]\!]^\circ &\overset{\text{def}}{=} \mathsf{false} \\
[\![\mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2]\!]^\circ &\overset{\text{def}}{=} \mathsf{if}\ [\![e]\!]^\circ\ \mathsf{then}\ [\![e_1]\!]^\circ\ \mathsf{else}\ [\![e_2]\!]^\circ \\
[\![\Sigma\, x : A.\, B]\!]^\circ &\overset{\text{def}}{=} \Sigma\, x : [\![A]\!]^\circ.\, [\![B]\!]^\circ \\
[\![\langle e_1, e_2 \rangle\ \mathbf{as}\ \Sigma\, x : A.\, B]\!]^\circ &\overset{\text{def}}{=} \langle [\![e_1]\!]^\circ, [\![e_2]\!]^\circ \rangle\ \mathsf{as}\ \Sigma\, x : [\![A]\!]^\circ.\, [\![B]\!]^\circ \\
[\![\mathbf{fst}\ e]\!]^\circ &\overset{\text{def}}{=} \mathsf{fst}\ [\![e]\!]^\circ \\
[\![\mathbf{snd}\ e]\!]^\circ &\overset{\text{def}}{=} \mathsf{snd}\ [\![e]\!]^\circ
\end{aligned}$$

Figure D.14: Model of ECC^{CC} in ECC^{D}

Note that the following ANF definitions for ECC^{CC} exclude dependent conditionals, since the ANF translation for dependent conditionals presented in Chapter 4 is incomplete.

$$
\begin{array}{llll}
\textit{Values} & \mathbf{V} & ::= & \mathbf{x} \mid \mathbf{U} \mid \mathbf{Code}\,(\mathbf{n}:\mathbf{M},\mathbf{x}:\mathbf{M}).\,\mathbf{M} \mid \langle\langle \mathbf{V},\mathbf{V}\rangle\rangle \\
 & & \mid & \mathbf{\Pi}\,\mathbf{x}:\mathbf{M}.\,\mathbf{M} \mid \boldsymbol{\lambda}\,(\mathbf{n}:\mathbf{M},\mathbf{x}:\mathbf{M}).\,\mathbf{M} \mid \mathbf{\Sigma}\,\mathbf{x}:\mathbf{M}.\,\mathbf{M} \\
 & & \mid & \langle \mathbf{V},\mathbf{V}\rangle\,\mathbf{as}\,\mathbf{M} \\
\textit{Computations} & \mathbf{N} & ::= & \mathbf{V} \mid \mathbf{V}\ \mathbf{V} \mid \mathbf{fst}\,\mathbf{V} \mid \mathbf{snd}\,\mathbf{V} \\
\textit{Configurations} & \mathbf{M} & ::= & \mathbf{N} \mid \mathbf{let}\,\mathbf{x}=\mathbf{N}\,\mathbf{in}\,\mathbf{M} \\
\textit{Continuations} & \mathbf{K} & ::= & [\cdot] \mid \mathbf{let}\,\mathbf{x}=[\cdot]\,\mathbf{in}\,\mathbf{M}
\end{array}
$$

Figure D.15: ECC^{CC} ANF

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

$$
\begin{array}{rcl}
\mathbf{K}\langle\langle\mathbf{N}\rangle\rangle & \overset{\text{def}}{=} & \mathbf{K}[\mathbf{N}] \\
\mathbf{K}\langle\langle\mathbf{let}\,\mathbf{x}=\mathbf{N}'\,\mathbf{in}\,\mathbf{M}\rangle\rangle & \overset{\text{def}}{=} & \mathbf{let}\,\mathbf{x}=\mathbf{N}'\,\mathbf{in}\,\mathbf{K}\langle\langle\mathbf{M}\rangle\rangle
\end{array}
$$

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

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

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

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

Figure D.16: ECC^{CC} Composition of Configurations

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

$$\begin{array}{rcl}
\mathbf{K}[\langle\langle(\boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{M}'_A, \mathbf{x} : \mathbf{M}_A).\,\mathbf{M}), \mathbf{V}'\rangle\rangle\ \mathbf{V}] & \mapsto_\beta & \mathbf{K}\langle\langle\mathbf{M}[\mathbf{V}'/\mathbf{n}][\mathbf{V}/\mathbf{x}]\rangle\rangle \\
\mathbf{K}[\mathbf{fst}\ \langle\mathbf{V}_1, \mathbf{V}_2\rangle] & \mapsto_{\pi_1} & \mathbf{K}[\mathbf{V}_1] \\
\mathbf{K}[\mathbf{snd}\ \langle\mathbf{V}_1, \mathbf{V}_2\rangle] & \mapsto_{\pi_2} & \mathbf{K}[\mathbf{V}_2] \\
\mathbf{let}\,\mathbf{x} = \mathbf{V}\,\mathbf{in}\,\mathbf{M} & \mapsto_\zeta & \mathbf{M}[\mathbf{V}/\mathbf{x}]
\end{array}$$

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

$$\frac{}{\vdash \mathbf{M} \mapsto^* \mathbf{M}}\ \textsc{RedA-Refl} \qquad\qquad \frac{\mathbf{M} \mapsto \mathbf{M}_1 \qquad \vdash \mathbf{M}_1 \mapsto^* \mathbf{M}'}{\vdash \mathbf{M} \mapsto^* \mathbf{M}'}\ \textsc{RedA-Trans}$$

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

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

Figure D.17: ECC^{CC} ANF Machine Evaluation

E REFERENCE FOR ABSTRACT CLOSURE CONVERSION

This appendix contains the complete definitions for the abstract closure conversion translation from Chapter 5. All figures related to the translation are reproduced and completed with elided parts here, and elided figures are presented here.

Typographical Note. *In this appendix, I typeset the source language, ECC^D, in a* blue, non-bold, sans-serif font, *and the target language, ECC^{CC}, in a* **bold, red, serif font**.

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

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

Figure E.1: Observation Relation between ECC^D and ECC^{CC}

$$\begin{aligned}
\mathrm{FV}(\mathsf{e},\mathsf{B},\mathsf{\Gamma}) \quad &\stackrel{\text{def}}{=} \quad \mathsf{\Gamma}_0,\dots,\mathsf{\Gamma}_n,(\mathsf{x}_0:\mathsf{A}_0,\dots,\mathsf{x}_n:\mathsf{A}_n) \\
&\textit{where} \quad \mathsf{x}_0,\dots,\mathsf{x}_n = \mathrm{fv}(\mathsf{e},\mathsf{B}) \\
&\qquad\quad \mathsf{\Gamma} \vdash \mathsf{x}_0 : \mathsf{A}_0 \\
&\qquad\quad \vdots \\
&\qquad\quad \mathsf{\Gamma} \vdash \mathsf{x}_n : \mathsf{A}_n \\
&\qquad\quad \mathsf{\Gamma}_0 = \mathrm{FV}(\mathsf{A}_0,\_,\mathsf{\Gamma}) \\
&\qquad\quad \vdots \\
&\qquad\quad \mathsf{\Gamma}_n = \mathrm{FV}(\mathsf{A}_n,\_,\mathsf{\Gamma})
\end{aligned}$$

Figure E.2: Dependent Free Variable Sequences

$\boxed{[\![\mathsf{e}]\!] = \mathbf{e}}$

$$[\![\mathsf{e}]\!] \quad = \quad \mathbf{e} \quad \text{where } \mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A} \text{ and } \mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow \mathbf{e}$$

$\boxed{[\![\mathsf{\Gamma}]\!] = \mathbf{\Gamma}}$

$$[\![\mathsf{\Gamma}]\!] \quad = \quad \mathbf{\Gamma} \quad \text{where } \vdash \mathsf{\Gamma} \text{ and } \vdash \mathsf{\Gamma} \rightsquigarrow \mathbf{\Gamma}$$

Figure E.3: Closure Conversion Syntactic Sugar

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

$$\frac{\Gamma \vdash A : U \rightsquigarrow \mathbf{A} \qquad \Gamma, x : A \vdash B : \mathsf{Prop} \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \mathsf{Prop} \rightsquigarrow \mathbf{\Pi\, x : A.\, B}}\ \text{CC-Pi-Prop}$$

$$\frac{\Gamma \vdash A : U \rightsquigarrow \mathbf{A} \qquad \Gamma, x : A \vdash B : \mathsf{Type}_i \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : \mathsf{Type}_i \rightsquigarrow \mathbf{\Pi\, x : A.\, B}}\ \text{CC-Pi-Type}$$

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

$$\frac{\Gamma \vdash e_1 : \Pi\, x : A.\, B \rightsquigarrow \mathbf{e}_1 \qquad \Gamma \vdash e_2 : A \rightsquigarrow \mathbf{e}_2}{\Gamma \vdash e_1\ e_2 : B[e_2/x] \rightsquigarrow \mathbf{e}_1\ \mathbf{e}_2}\ \text{CC-App}$$

Figure E.4: Abstract Closure Conversion from ECC^D to ECC^{CC} (1/2)

$\boxed{\Gamma \vdash \mathsf{e} : \mathsf{t} \rightsquigarrow \mathbf{e} \quad \text{where } \Gamma \vdash \mathsf{e} : \mathsf{t}}$

$$\frac{}{\Gamma \vdash \mathsf{x} : \mathsf{A} \rightsquigarrow \mathbf{x}}\ \textsc{CC-Var} \qquad \frac{}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_1 \rightsquigarrow \mathbf{Prop}}\ \textsc{CC-Prop}$$

$$\frac{}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1} \rightsquigarrow \mathbf{Type}_{i+1}}\ \textsc{CC-Type}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}_i \rightsquigarrow \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{Type}_i \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} : \mathsf{Type}_i \rightsquigarrow \mathbf{\Sigma\, x : A.\, B}}\ \textsc{CC-Sig}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{A} \rightsquigarrow \mathbf{e}_1 \\ \Gamma \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}] \rightsquigarrow \mathbf{e}_2 \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type}_i \rightsquigarrow \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{Type}_i \rightsquigarrow \mathbf{B}\end{array}}{\Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle\ \mathsf{as}\ \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\ \mathbf{\Sigma\, x : A.\, B}}\ \textsc{CC-Pair}$$

$$\frac{\Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow \mathbf{e}}{\Gamma \vdash \mathsf{fst}\ \mathsf{e} : \mathsf{A} \rightsquigarrow \mathbf{fst\ e}}\ \textsc{CC-Fst} \qquad \frac{\Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow \mathbf{e}}{\Gamma \vdash \mathsf{snd}\ \mathsf{e} : \mathsf{B}[\mathsf{fst}\ \mathsf{e}/\mathsf{x}] \rightsquigarrow \mathbf{snd\ e}}\ \textsc{CC-Snd}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \rightsquigarrow \mathbf{true}}\ \textsc{CC-True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \rightsquigarrow \mathbf{false}}\ \textsc{CC-False}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \rightsquigarrow \mathbf{e} \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{B}[\mathsf{true}/\mathsf{y}] \rightsquigarrow \mathbf{e}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{false}/\mathsf{y}] \rightsquigarrow \mathbf{e}_2}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \mathsf{B}[\mathsf{e}/\mathsf{y}] \rightsquigarrow \mathbf{if\ e\ then\ e_1\ else\ e_2}}\ \textsc{CC-If}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow \mathbf{e} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{e}' : \mathsf{B} \rightsquigarrow \mathbf{e}'}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}' : \mathsf{B}[\mathsf{e}/\mathsf{x}] \rightsquigarrow \mathbf{let\ x = e\ in\ e'}}\ \textsc{CC-Let} \qquad \frac{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow \mathbf{e}}{\Gamma \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow \mathbf{e}}\ \textsc{CC-Conv}$$

$\boxed{\vdash \Gamma \rightsquigarrow \mathbf{\Gamma} \quad \text{where } \vdash \Gamma}$

$$\frac{}{\vdash \cdot \rightsquigarrow \cdot}\ \textsc{CC-Empty} \qquad \frac{\vdash \Gamma \rightsquigarrow \mathbf{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{U} \rightsquigarrow \mathbf{A}}{\vdash \Gamma, \mathsf{x} : \mathsf{A} \rightsquigarrow \mathbf{\Gamma, x : A}}\ \textsc{CC-Assum}$$

$$\frac{\vdash \Gamma \rightsquigarrow \mathbf{\Gamma} \qquad \Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow \mathbf{e}}{\vdash \Gamma, \mathsf{x} = \mathsf{e} \rightsquigarrow \mathbf{\Gamma, x = e}}\ \textsc{CC-Def}$$

Figure E.5: Abstract Closure Conversion from ECC^{D} to ECC^{CC} (2/2)

F | REFERENCE FOR COC^D

This appendix contains the complete definitions for CoC^D. All CoC^D figures from Chapter 6 are reproduced and completed with elided parts here, and elided figures are presented here. These figures contained extensions with definitions required to fully formalize the compiler correctness results, including booleans with non-dependent elimination.

Typographical Note. *In this appendix, I typeset CoC^D in a* blue, non-bold, sans-serif font.

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

Figure F.1: CoC^D Syntax

$$
\begin{array}{llcl}
\textit{Kinds} & \mathsf{K} & ::= & \star \mid \Pi\,\alpha : \mathsf{K.\, K} \mid \Pi\,\mathsf{x : A.\, K} \\
\textit{Types} & \mathsf{A, B} & ::= & \alpha \mid \Pi\,\mathsf{x : A.\, B} \mid \Pi\,\alpha : \mathsf{K.\, B} \mid \lambda\,\mathsf{x : A.\, B} \mid \lambda\,\alpha : \mathsf{K.\, B} \mid \mathsf{A\ e} \\
 & & \mid & \mathsf{A\ B} \mid \Sigma\,\mathsf{x : A.\, B} \mid \mathsf{bool} \mid \mathsf{let}\ \alpha = \mathsf{A : K\ in\ B} \\
 & & \mid & \mathsf{let\ x = e : A\ in\ B} \\
\textit{Terms} & \mathsf{e} & ::= & \mathsf{x} \mid \lambda\,\mathsf{x : A.\, e} \mid \lambda\,\alpha : \mathsf{K.\, e} \mid \mathsf{e\ e} \mid \mathsf{e\ A} \mid \langle \mathsf{e_1, e_2} \rangle\ \mathsf{as}\ \Sigma\,\mathsf{x : A.\, B} \\
 & & \mid & \mathsf{fst\ e} \mid \mathsf{snd\ e} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if\ e\ then\ e_1\ else\ e_2} \\
 & & \mid & \mathsf{let\ x = e : A\ in\ e} \mid \mathsf{let}\ \alpha = \mathsf{A : K\ in\ e} \\
\textit{Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x : A} \mid \Gamma, \mathsf{x = e : A,} \mid \Gamma, \alpha : \mathsf{K} \mid \Gamma, \alpha = \mathsf{A : K}
\end{array}
$$

Figure F.2: CoC^D Explicit Syntax

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

$$\begin{array}{rll} \Gamma \vdash (\lambda\, x : A.\, e_1)\; e_2 & \rhd_\beta & e_1[e_2/x] \\ \Gamma \vdash \mathsf{fst}\; \langle e_1, e_2 \rangle & \rhd_{\pi_1} & e_1 \\ \Gamma \vdash \mathsf{snd}\; \langle e_1, e_2 \rangle & \rhd_{\pi_2} & e_2 \\ \Gamma \vdash \mathsf{if\ true\ then}\; e_1\; \mathsf{else}\; e_2 & \rhd_{\iota_1} & e_1 \\ \Gamma \vdash \mathsf{if\ false\ then}\; e_1\; \mathsf{else}\; e_2 & \rhd_{\iota_1} & e_2 \\ \Gamma \vdash x & \rhd_\delta & e \qquad\qquad \text{where } x = e : A \in \Gamma \\ \Gamma \vdash \mathsf{let}\; x = e_2 : A\; \mathsf{in}\; e_1 & \rhd_\zeta & e_1[e_2/x] \end{array}$$

Figure F.3: CoC^D Reduction

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash A \rhd^* A' \qquad \Gamma, x = e' : A' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \text{let } x = e_1 : A \text{ in } e_2 \rhd^* \text{let } x = e_1' : A' \text{ in } e_2'}\ \textsc{Red-Cong-Let}$$

Figure F.4: CoC^D Conversion

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

$$\frac{\Gamma \vdash e \triangleright^* e_1 \qquad \Gamma \vdash e' \triangleright^* e_1}{\Gamma \vdash e \equiv e'}\ \equiv$$

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

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

Figure F.5: CoC^D Equivalence

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

$$\frac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \star : \square}\ * \qquad \frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi\, x : A.\, B : \star}\ \textsc{Pi-}*$$

$$\frac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi\, x : A.\, B : \square}\ \textsc{Pi-}\square \qquad \frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash \Pi\, x : A.\, B : U}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B}\ \textsc{Lam}$$

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

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

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : B \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B}\ \textsc{If}$$

$$\frac{\Gamma \vdash e' : A \qquad \Gamma, x = e' : A \vdash e : B}{\Gamma \vdash \mathsf{let}\, x = e' : A\, \mathsf{in}\, e : B[e'/x]}\ \textsc{Let} \qquad \frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : B}\ \textsc{Conv}$$

Figure F.6: CoC^D Typing

$\boxed{\vdash \Gamma}$

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

$$\frac{\vdash \Gamma \qquad \Gamma \vdash e : A \qquad \Gamma \vdash A : U}{\vdash \Gamma, x = e : A}\ \textsc{W-Def}$$

Figure F.7: CoC^D Well-Formed Environments

$$\textit{CoC}^D\ \textit{Observations} \quad v \quad ::= \quad \mathsf{true} \mid \mathsf{false}$$

Figure F.8: CoC^D Observations

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

$$\frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash e} \qquad \frac{\cdot \vdash e}{\vdash e}$$

Figure F.9: CoC^D Components and Programs

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

$$\mathsf{eval}(e) \quad = \quad v \quad \text{if} \vdash e \text{ and } \cdot \vdash e \rhd^* v$$

Figure F.10: CoC^D Evaluation

$$\textit{Closing Substitutions} \quad \gamma \quad \overset{\text{def}}{=} \quad \cdot \mid \gamma[x \mapsto e]$$

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

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

$\boxed{\gamma(e) = e}$

$$\cdot(e) = e \qquad \gamma[x \mapsto e'](e) = \gamma(e[x/e'])$$

Figure F.11: CoC^D Closing Substitutions and Linking

G REFERENCE FOR COC^k

This appendix contains the complete definitions for CoC^k. All CoC^k figures from Chapter 6 are reproduced and completed with elided parts here, and elided figures are presented here.

Typographical Note. *In this appendix, I typeset CoC^k in a* **bold, red, serif font**.

$$
\begin{array}{llcl}
\textit{Universes} & \mathbf{U} & ::= & \star \mid \square \\
\textit{Expressions} & \mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \star \mid \mathbf{\Pi\, x : A.\, e} \mid \mathbf{\lambda\, x : A.\, e} \mid \mathbf{e\ e} \mid \mathbf{e\ @\ A\ e} \\
 & & \mid & \mathbf{\Sigma\, x : A.\, B} \mid \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\ \mathbf{\Sigma\, x : A.\, B} \mid \mathbf{fst\ e} \mid \mathbf{snd\ e} \\
 & & \mid & \mathbf{bool} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if\ e\ then\ e_1\ else\ e_2} \\
 & & \mid & \mathbf{let\ x = e : A\ in\ e} \\
\textit{Environments} & \mathbf{\Gamma} & ::= & \cdot \mid \mathbf{\Gamma, x : A} \mid \mathbf{\Gamma, x = e : A}
\end{array}
$$

Figure G.1: CoC^k Syntax

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

$$
\begin{array}{rll}
\mathbf{\Gamma} \vdash (\mathbf{\lambda\, x : A.\, e_1})\ \mathbf{e_2} & \rhd_\beta & \mathbf{e_1}[\mathbf{e_2}/\mathbf{x}] \\
\mathbf{\Gamma} \vdash \mathbf{\lambda\, \alpha : \star.\, e_1\ @\ A\ e_2} & \rhd_@ & (\mathbf{e_1}[\mathbf{A}/\mathbf{\alpha}])\ \mathbf{e_2} \\
\mathbf{\Gamma} \vdash \mathbf{fst}\ \langle \mathbf{e_1}, \mathbf{e_2} \rangle & \rhd_{\pi_1} & \mathbf{e_1} \\
\mathbf{\Gamma} \vdash \mathbf{snd}\ \langle \mathbf{e_1}, \mathbf{e_2} \rangle & \rhd_{\pi_2} & \mathbf{e_2} \\
\mathbf{\Gamma} \vdash \mathbf{if\ true\ then\ e_1\ else\ e_2} & \rhd_{\iota_1} & \mathbf{e_1} \\
\mathbf{\Gamma} \vdash \mathbf{if\ false\ then\ e_1\ else\ e_2} & \rhd_{\iota_1} & \mathbf{e_2} \\
\mathbf{\Gamma} \vdash \mathbf{x} & \rhd_\delta & \mathbf{e} \qquad\qquad \text{where } \mathbf{x = e} \in \mathbf{\Gamma} \\
\mathbf{\Gamma} \vdash \mathbf{let\ x = e_2 : A\ in\ e_1} & \rhd_\zeta & \mathbf{e_1}[\mathbf{e_2}/\mathbf{x}]
\end{array}
$$

Figure G.2: CoC^k Reduction

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

$$\frac{\Gamma \vdash e \rhd^* e_1 \qquad \Gamma \vdash e' \rhd^* e_1}{\Gamma \vdash e \equiv e'}\ \equiv$$

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

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

$$\frac{}{\Gamma \vdash (e_1\ @\ A\ (\lambda\, x : B.\, e_2)) \equiv (\lambda\, x : B.\, e_2)\ (e_1\ B\ \mathbf{id})}\ \equiv\text{-}\textsc{Cont} \qquad \frac{\Gamma \vdash e' \equiv e}{\Gamma \vdash e \equiv e'}\ \equiv\text{-}\textsc{Sym}$$

$$\frac{\Gamma \vdash e \equiv e_1 \qquad \Gamma \vdash e_1 \equiv e'}{\Gamma \vdash e \equiv e'}\ \equiv\text{-}\textsc{Trans}$$

Figure G.3: CoC^k Equivalence

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash A \rhd^* A' \qquad \Gamma \vdash e_2 \rhd^* e_2'}{\Gamma \vdash e_1\ @\ A\ e_2 \rhd^* e_1'\ @\ A'\ e_2'}\ \textsc{Red-Cong-Cont}$$

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

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

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

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash A \rhd^* A' \qquad \Gamma, x = e' : A' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \text{let}\, x = e_1 : A\, \text{in}\, e_2 \rhd^* \text{let}\, x = e_1' : A'\, \text{in}\, e_2'}\ \textsc{Red-Cong-Let}$$

Figure G.4: CoC^k Conversion

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

$$\frac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \star : \square}\ *$$

$$\frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi\, x : A.\, B : \star}\ \textsc{Pi-}* \qquad \frac{\Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi\, x : A.\, B : \square}\ \textsc{Pi-}\square$$

$$\frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash \Pi\, x : A.\, B : U}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B}\ \textsc{Lam}$$

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

$$\frac{\Gamma \vdash e : \Pi\, \alpha : \star.\, (B \to \alpha) \to \alpha \qquad \Gamma \vdash A : \star \qquad \Gamma, x = e\ B\ \mathsf{id} \vdash e' : A}{\Gamma \vdash e\ @\ A\ (\lambda\, x : B.\, e') : A}\ \textsc{T-Cont}$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma\, x : A.\, B : \star}\ \textsc{Sig} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash \langle e_1, e_2 \rangle \text{ as } \Sigma\, x : A.\, B : \Sigma\, x : A.\, B}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{fst}\, e : A}\ \textsc{Fst} \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x]}\ \textsc{Snd} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{bool} : \star}\ \textsc{Bool}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \textsc{True} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : B \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B}\ \textsc{If}$$

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

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : B}\ \textsc{Conv}$$

Figure G.5: CoC^k Typing

$\boxed{\vdash \Gamma}$

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

$$\frac{\vdash \Gamma \qquad \Gamma \vdash e : A \qquad \Gamma \vdash A : U}{\vdash \Gamma, x = e : A}\ \textsc{W-Def}$$

Figure G.6: CoC^k Well-formed Environments

$$\textit{CoC}^k\ \textit{Observations} \quad v \quad ::= \quad \mathbf{true} \mid \mathbf{false}$$

Figure G.7: CoC^k Observations

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

$$\frac{\Gamma \vdash e : \Pi\, \alpha : \mathbf{Prop}\,.\,(\mathbf{bool} \to \alpha) \to \alpha}{\Gamma \vdash e} \qquad \frac{\cdot \vdash e}{\vdash e}$$

Figure G.8: CoC^k Components and Programs

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

$$\mathrm{eval}(e) \quad = \quad v \quad \text{if} \vdash e \text{ and } \cdot \vdash e\ @\ \mathbf{bool}\ \mathbf{id} \rhd^* v$$

Figure G.9: CoC^k Evaluation

$$\textit{Closing Substitutions} \quad \gamma \quad \overset{\text{def}}{=} \quad \cdot \mid \gamma[x \mapsto e]$$

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

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

$\boxed{\gamma(e) = e}$

$$\cdot(e) = e \qquad\qquad \gamma[x \mapsto e'](e) = \gamma(e[x/e'])$$

Figure G.10: CoC^k Closing Substitutions and Linking

$\boxed{\Gamma \vdash e_1 \equiv e_2}$

All other rules identical to CoC^{D}

$$\frac{\Gamma \vdash p : e_1 = e_2}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-}\textsc{Ext}$$

Figure G.11: Extensional CoC Equivalence

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

All other rules are homomorphic

$$\frac{\begin{array}{c}\mathbf{\Gamma \vdash e} : \_ \rightsquigarrow_{\circ} e \\ \mathbf{\Gamma \vdash B} : \_ \rightsquigarrow_{\circ} B \qquad \mathbf{\Gamma \vdash A} : \_ \rightsquigarrow_{\circ} A \qquad \mathbf{\Gamma, x = e\ B\ id \vdash e'} : \mathbf{A} \rightsquigarrow_{\circ} e'\end{array}}{\mathbf{\Gamma \vdash e\ @\ A\ (\lambda\, x : B.\, e') : A} \rightsquigarrow_{\circ} let\, x = e\ B\ id : B\, in\, e'}\ \textsc{Un-Cont}$$

Figure G.12: Model of CoC^{k} in Extensional CoC

```
Declare ML Module "paramcoq".

Variable A : Prop.
Variable B : Prop.

Variable R_b : B -> B -> Prop.

Realizer A as A_r arity 2 := (fun a1 a2 => a1 = a2).
Realizer B as B_r arity 2 := R_b.

Definition cpsT := forall (α:Prop), (B -> α) -> α.

Parametricity cpsT.

Definition id := fun (x : B) => x.

Definition kT := (B -> A).

Parametricity kT.

(* Note that by the fundamental property, the premises are trivially
   satisfied, but the translation really wants an honest-to-goodness
   term and not a variable. *)
Theorem Cont_Shuffle : forall e k, (cpsT_R e e) -> (kT_R k k) ->
                              ((e A k) = (k (e B id))).
Proof.
  intros e k P_e P_k.
  unfold cpsT_R in P_e.
  (* The only clever part: instantiate the relation between types
     A and B to be that a is related to (k b) in the A relation*)
  set (P := (P_e A B (fun (a:A) (b:B) => (A_r a (k b))) k id)).
  unfold id in P at 1.

  exact (P (fun b1 b2 H0 => (P_k b1 b2 H0))).
Qed.
```

Listing G.1: Coq proof of Lemma 6.4.2

H REFERENCE FOR CPS^n

This appendix contains the complete definitions for the CBN CPS translation from CoC^D to CoC^k. All translation figures from Chapter 6 are reproduced and completed with elided parts here, and elided figures are presented here.

Typographical Note. *In this appendix, I typeset the source language, CoC^D, in a* blue, non-bold, sans-serif font, *and the target language, CoC^k, in a* **bold, red, serif font**.

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

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

Figure H.1: Observation Relation between CoC^D and CoC^k

$\boxed{\Gamma \vdash \mathsf{U} \rightsquigarrow^n_U \mathbf{U}}$

$$\frac{}{\Gamma \vdash \star \rightsquigarrow^n_U \boldsymbol{\star}}\ \text{CPS}^n_U\text{-Star} \qquad\qquad \frac{}{\Gamma \vdash \Box \rightsquigarrow^n_U \boldsymbol{\Box}}\ \text{CPS}^n_U\text{-Box}$$

Figure H.2: CPS^n of Universes

$\boxed{\Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash \mathsf{K}^+ : \mathsf{U}^+}$

$$\frac{}{\Gamma \vdash \star : \Box \rightsquigarrow^n_\kappa \boldsymbol{\star}}\ \text{CPS}^n_\kappa\text{-Ax} \qquad \frac{\Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : \mathsf{K} \vdash \mathsf{K}' : \mathsf{U}' \rightsquigarrow^n_\kappa \boldsymbol{\kappa}'}{\Gamma \vdash \Pi\, \alpha : \mathsf{K}.\, \mathsf{K}' : \mathsf{U}' \rightsquigarrow^n_\kappa \boldsymbol{\Pi\, \alpha : \kappa.\, \kappa'}}\ \text{CPS}^n_\kappa\text{-PiK}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\Gamma \vdash \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\Pi\, \mathbf{x} : \mathbf{A}.\, \kappa}}\ \text{CPS}^n_\kappa\text{-PiA}$$

Figure H.3: CPS^n of Kinds

$\boxed{\Gamma \vdash A : K \rightsquigarrow^n_A \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash A^+ : K^+}$

$$\frac{}{\Gamma \vdash \alpha : K \rightsquigarrow^n_A \boldsymbol{\alpha}}\ \text{CPS}^n_A\text{-Var}$$

$$\frac{\Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : K' \rightsquigarrow^n_{A\div} \mathbf{B}}{\Gamma \vdash \Pi x : A.\, B : K' \rightsquigarrow^n_A \boldsymbol{\Pi}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ \text{CPS}^n_A\text{-Pi}$$

$$\frac{\Gamma \vdash K : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, x : A \vdash B : K' \rightsquigarrow^n_{A\div} \mathbf{B}}{\Gamma \vdash \Pi \alpha : K.\, B : K' \rightsquigarrow^n_A \boldsymbol{\Pi}\, \boldsymbol{\alpha} : \boldsymbol{\kappa}.\, \mathbf{B}}\ \text{CPS}^n_A\text{-PiK}$$

$$\frac{\Gamma \vdash A : K' \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : K \rightsquigarrow^n_A \mathbf{B}}{\Gamma \vdash \lambda x : A.\, B : \Pi x : A.\, K \rightsquigarrow^n_A \boldsymbol{\lambda}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ \text{CPS}^n_A\text{-Constr}$$

$$\frac{\Gamma \vdash K : U \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : K \vdash B : K' \rightsquigarrow^n_A \mathbf{B}}{\Gamma \vdash \lambda \alpha : K.\, B : \Pi \alpha : K.\, K' \rightsquigarrow^n_A \boldsymbol{\lambda}\, \boldsymbol{\alpha} : \boldsymbol{\kappa}.\, \mathbf{B}}\ \text{CPS}^n_A\text{-Abs}$$

$$\frac{\Gamma \vdash A : \Pi x : B.\, K \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash e : B \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash A\ e : K[e/x] \rightsquigarrow^n_A \mathbf{A}\ \mathbf{e}}\ \text{CPS}^n_A\text{-AppConstr}$$

$$\frac{\Gamma \vdash A : \Pi \alpha : K'.\, K \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash B : K' \rightsquigarrow^n_A \mathbf{B}}{\Gamma \vdash A\ B : K[B/\alpha] \rightsquigarrow^n_A \mathbf{A}\ \mathbf{B}}\ \text{CPS}^n_A\text{-Inst}$$

$$\frac{\Gamma \vdash A : \star \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow^n_{A\div} \mathbf{B}}{\Gamma \vdash \Sigma x : A.\, B : \star \rightsquigarrow^n_A \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ \text{CPS}^n_A\text{-Sig}$$

$$\frac{}{\Gamma \vdash \mathsf{bool} : \star \rightsquigarrow^n_A \mathbf{bool}}\ \text{CPS}^n_A\text{-Bool}$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow^n_e \mathbf{e} \qquad \Gamma \vdash A : K \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, x = e : A \vdash B : K' \rightsquigarrow^n_A \mathbf{B}}{\Gamma \vdash \mathsf{let}\, x = e : A\, \mathsf{in}\, B : K' \rightsquigarrow^n_A \mathbf{let}\, \mathbf{x} = \mathbf{e} : \mathbf{A}\, \mathbf{in}\, \mathbf{B}}\ \text{CPS}^n_A\text{-Let} \qquad \ldots$$

Figure H.4: CPS^n of Types (1/2)

$\boxed{\Gamma \vdash A : K \leadsto^n_A \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash A^+ : K^+}$

$$\cdots \qquad \frac{\vdash \Gamma \leadsto^n \mathbf{\Gamma} \qquad \Gamma \vdash A : K \leadsto^n_\kappa \boldsymbol{\kappa} \qquad \begin{array}{c}\Gamma \vdash A : K \leadsto^n_A \mathbf{A} \\ \Gamma, \alpha = A : K \vdash B : \_ \leadsto^n_A \mathbf{B}\end{array}}{\Gamma \vdash \mathsf{let}\, \alpha = A : K \,\mathsf{in}\, B : \_ \leadsto^n_A \mathbf{let}\, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}\, \mathbf{in}\, \mathbf{B}} \text{ CPS}^n_A\text{-LetK}$$

$$\frac{\Gamma \vdash A : K' \qquad \Gamma \vdash K \equiv K' \qquad \Gamma \vdash A : K' \leadsto^n_A \mathbf{A}}{\Gamma \vdash A : K \leadsto^n_A \mathbf{A}} \text{ CPS}^n_A\text{-Conv}$$

$\boxed{\Gamma \vdash A : \star \leadsto^n_{A^\div} \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash A^\div : \star^+}$

$$\frac{\Gamma \vdash A : \star \leadsto^n_A \mathbf{A}}{\Gamma \vdash A : \star \leadsto^n_{A^\div} \mathbf{\Pi}\, \boldsymbol{\alpha} : \boldsymbol{\star}.\, (\mathbf{A} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}} \text{ CPS}^n_{A^\div}\text{-Comp}$$

Figure H.5: CPS^n of Types (1/2)

$$\boxed{\Gamma \vdash e : A \leadsto^n_e \mathbf{A} \text{ Lemma 6.5.5 will show } \Gamma^+ \vdash e^{\div} : A^{\div}}$$

$$\frac{\Gamma \vdash A : K \leadsto^n_A \mathbf{A}}{\Gamma \vdash x : A \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : A \to \alpha.\, x\ \alpha\ k}}\ \text{CPS}^n_e\text{-Var}$$

$$\frac{\Gamma \vdash A : K \leadsto^n_{A\div} \mathbf{A} \qquad \Gamma, x : A \vdash B : K' \leadsto^n_{A\div} \mathbf{B} \qquad \Gamma, x : A \vdash e : B \leadsto^n_e \mathbf{e}}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : (\Pi\, x : A.\, B) \to \alpha.\, k\ (\lambda\, x : A.\, e)}}\ \text{CPS}^n_e\text{-Fun}$$

$$\frac{\Gamma \vdash K : \_ \leadsto^n_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : K \vdash B : \_ \leadsto^n_{A\div} \mathbf{B} \qquad \Gamma, \alpha : K \vdash e : B \leadsto^n_e \mathbf{e}}{\begin{array}{l}\Gamma \vdash \lambda\, \alpha : K.\, e : \Pi\, \alpha : K.\, B \leadsto^n_e \boldsymbol{\lambda\, \alpha_{ans} : \star.\, \lambda\, k : (\Pi\, \alpha : \kappa.\, B) \to \alpha_{ans}.} \\ \qquad\qquad\qquad\qquad \boldsymbol{k\ (\lambda\, \alpha : \kappa.\, e)}\end{array}}\ \text{CPS}^n_e\text{-Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \Pi\, x : A.\, B \leadsto^n_e \mathbf{e} \qquad \Gamma \vdash A : K \leadsto^n_{A\div} \mathbf{A}^{\div} \\ \Gamma, x : A \vdash B : K' \leadsto^n_{A\div} \mathbf{B}^{\div} \qquad \Gamma, x : A \vdash B : K' \leadsto^n_A \mathbf{B}^{+} \qquad \Gamma \vdash e' : A \leadsto^n_e \mathbf{e}'\end{array}}{\begin{array}{l}\Gamma \vdash e\ e' : B[e'/x] \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : (B^{+}[e'/x]) \to \alpha.} \\ \qquad\qquad\qquad \boldsymbol{e\ \alpha\ (\lambda\, f : \Pi\, x : A^{\div}.\, B^{\div}.\, (f\ e')\ \alpha\ k)}\end{array}}\ \text{CPS}^n_e\text{-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \Pi\, \alpha : K.\, B \leadsto^n_e \mathbf{e} \\ \Gamma, \alpha : K \vdash B : \_ \leadsto^n_{A\div} \mathbf{B}^{\div} \qquad \Gamma, \alpha : K \vdash B : \_ \leadsto^n_A \mathbf{B}^{+} \qquad \Gamma \vdash A : K \leadsto^n_A \mathbf{A}\end{array}}{\begin{array}{l}\Gamma \vdash e\ A : B[A/\alpha] \leadsto^n_e \boldsymbol{\lambda\, \alpha_{ans} : \star.\, \lambda\, k : (B^{+}[A/\alpha]) \to \alpha_{ans}.} \\ \qquad\qquad\qquad \boldsymbol{e\ \alpha\ (\lambda\, f : \Pi\, \alpha : \kappa.\, B^{\div}.\, (f\ A)\ \alpha_{ans}\ k)}\end{array}}\ \text{CPS}^n_e\text{-Inst}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : bool \to \alpha.\, k\ true}}\ \text{CPS}^n_e\text{-True}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : bool \to \alpha.\, k\ false}}\ \text{CPS}^n_e\text{-False}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{bool} \leadsto^n_e \mathbf{e} \\ \Gamma \vdash B : \star \leadsto^n_A \mathbf{B} \qquad \Gamma \vdash e_1 : B \leadsto^n_e \mathbf{e}_1 \qquad \Gamma \vdash e_2 : B \leadsto^n_e \mathbf{e}_2\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : B \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : B \to \alpha.} \\ \qquad\qquad \boldsymbol{e\ @\ \alpha\ (\lambda\, x : bool.\, if\ x\ then\ (e_1\ @\ \alpha\ k)} \\ \qquad\qquad\qquad\qquad \boldsymbol{else\ (e_2\ @\ \alpha\ k))}\end{array}}\ \text{CPS}^n_e\text{-If}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : A \leadsto^n_e \mathbf{e} \qquad \Gamma \vdash A : K \leadsto^n_{A\div} \mathbf{A} \\ \Gamma, x = e : A \vdash B : K' \leadsto^n_A \mathbf{B} \qquad \Gamma, x = e : A \vdash e' : B \leadsto^n_e \mathbf{e}'\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\ x = e : A\ \mathsf{in}\ e' : B[e/x] \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : B[e/x] \to \alpha.} \\ \qquad\qquad \boldsymbol{let\ x = e : A\ in\ e'\ \alpha\ k}\end{array}}\ \text{CPS}^n_e\text{-Let}$$

Figure H.6: CPS^n of Terms (1/2)

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\kappa} \\ \Gamma, \alpha = \mathsf{A} : \mathsf{K} \vdash \mathsf{B} : \mathsf{K}' \rightsquigarrow^n_A \mathbf{B} \qquad \Gamma, \alpha = \mathsf{A} : \mathsf{K} \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\, \alpha = \mathsf{A} : \mathsf{K}\, \mathsf{in}\, \mathsf{e} : \mathsf{B}[\mathsf{A}/\alpha] \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha}_{ans} : \star.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{B}[\mathbf{A}/\mathbf{x}] \to \boldsymbol{\alpha}_{ans}. \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}\, \mathbf{in}\, \mathbf{e}\ \boldsymbol{\alpha}_{ans}\ \mathbf{k}\end{array}}\ \text{CPS}^n_e\text{-LetK}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^n_e \mathbf{e}}{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^n_e \mathbf{e}}\ \text{CPS}^n_e\text{-Conv}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{A} \rightsquigarrow^n_e \mathbf{e}_1 \\ \Gamma \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}] \rightsquigarrow^n_e \mathbf{e}_2 \qquad \Gamma \vdash \mathsf{A} : \star \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^n_{A\div} \mathbf{B}\end{array}}{\begin{array}{l}\Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : \star.\, \boldsymbol{\lambda}\, \mathbf{k} : \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B} \to \boldsymbol{\alpha}. \\ \qquad\qquad\qquad\qquad\qquad \mathbf{k}\ \langle \mathbf{e}_1, \mathbf{e}_2 \rangle\ \mathbf{as}\, \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}\end{array}}\ \text{CPS}^n_e\text{-Pair}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \star \rightsquigarrow^n_{A\div} \mathbf{A}^{\div} \\ \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^n_{A\div} \mathbf{B}^{\div} \qquad \Gamma \vdash \mathsf{A} : \star \rightsquigarrow^n_A \mathbf{A}^{+} \qquad \Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{fst}\, \mathsf{e} : \mathsf{A} \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : \star.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{A}^{+} \to \boldsymbol{\alpha}. \\ \qquad\qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}^{\div}.\, \mathbf{B}^{\div}.\, \mathbf{let}\, \mathbf{z} = \mathbf{fst}\, \mathbf{y}\, \mathbf{in}\, \mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k})\end{array}}\ \text{CPS}^n_e\text{-Fst}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \star \rightsquigarrow^n_{A\div} \mathbf{A}^{\div} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^n_{A\div} \mathbf{B}^{\div} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^n_A \mathbf{B}^{+} \\ \Gamma \vdash (\mathsf{fst}\, \mathsf{e}) : \mathsf{A} \rightsquigarrow^n_e (\mathbf{fst}\, \mathbf{e})^{\div} \qquad \Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^n_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{snd}\, \mathsf{e} : \mathsf{B}[(\mathsf{fst}\, \mathsf{e})/\mathsf{x}] \rightsquigarrow^n_e \boldsymbol{\lambda}\, \boldsymbol{\alpha} : \star.\, \boldsymbol{\lambda}\, \mathbf{k} : \mathbf{B}^{+}[(\mathbf{fst}\, \mathbf{e})^{\div}/\mathbf{x}] \to \boldsymbol{\alpha}. \\ \qquad\qquad\qquad\qquad \mathbf{e}\ @\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda}\, \mathbf{y} : \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}^{\div}.\, \mathbf{B}^{\div}. \\ \qquad\qquad\qquad\qquad\qquad \mathbf{let}\, \mathbf{z} = \mathbf{snd}\, \mathbf{y}\, \mathbf{in}\, \mathbf{z}\ \boldsymbol{\alpha}\ \mathbf{k})\end{array}}\ \text{CPS}^n_e\text{-Snd}$$

Figure H.7: CPS^n of Terms (2/2)

$\boxed{\vdash \Gamma \rightsquigarrow^n \boldsymbol{\Gamma}\ \text{Lemma 6.5.5 will show} \vdash \boldsymbol{\Gamma}^{+}}$

$$\frac{}{\vdash \cdot \rightsquigarrow^n \cdot}\ \text{CPS}^n_\Gamma\text{-Empty} \qquad \frac{\vdash \Gamma \rightsquigarrow^n \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^n_{A\div} \mathbf{A}}{\vdash \Gamma, \mathsf{x} : \mathsf{A} \rightsquigarrow^n \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A}}\ \text{CPS}^n_\Gamma\text{-AssumT}$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha : \mathsf{K} \rightsquigarrow^n \boldsymbol{\Gamma}, \boldsymbol{\alpha} : \boldsymbol{\kappa}}\ \text{CPS}^n_\Gamma\text{-AssumK}$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^n_{A\div} \mathbf{A} \qquad \Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^n_e \mathbf{e}}{\vdash \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A} \rightsquigarrow^n \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}}\ \text{CPS}^n_\Gamma\text{-Def}$$

$$\frac{\vdash \Gamma \rightsquigarrow^n \boldsymbol{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^n_A \mathbf{A} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^n_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha = \mathsf{A} : \mathsf{K} \rightsquigarrow^n \boldsymbol{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}}\ \text{CPS}^n_\Gamma\text{-DefT}$$

Figure H.8: CPS^n of Environments

I REFERENCE FOR CPS^v

This appendix contains the complete definitions for the CBV CPS translation from CoC^D to CoC^k. All translation figures from Chapter 6 are reproduced and completed with elided parts here, and elided figures are presented here.

Typographical Note. *In this appendix, I typeset the source language, CoC^D, in a* blue, non-bold, sans-serif font, *and the target language, CoC^k, in a* **bold, red, serif font**.

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

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

Figure I.1: Observation Relation between CoC^D and CoC^k

$\boxed{\Gamma \vdash \mathsf{U} \leadsto^v_U \mathbf{U}}$

$$\frac{}{\Gamma \vdash \star \leadsto^v_U \boldsymbol{\star}}\ \text{CPS}^v_U\text{-Star} \qquad\qquad \frac{}{\Gamma \vdash \square \leadsto^v_U \boldsymbol{\square}}\ \text{CPS}^v_U\text{-Box}$$

Figure I.2: CPS^v of Universes

$\boxed{\Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^v_\kappa \boldsymbol{\kappa} \text{ Lemma 6.6.6 will show } \Gamma^+ \vdash \mathsf{K}^+ : \mathsf{U}^+}$

$$\frac{}{\Gamma \vdash \star : \square \leadsto^v_\kappa \boldsymbol{\star}}\ \text{CPS}^v_\kappa\text{-Ax} \qquad \frac{\Gamma \vdash \mathsf{K} : \mathsf{U} \leadsto^v_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : \mathsf{K} \vdash \mathsf{K}' : \mathsf{U} \leadsto^v_\kappa \boldsymbol{\kappa}'}{\Gamma \vdash \Pi\, \alpha : \mathsf{K}.\, \mathsf{K}' : \mathsf{U} \leadsto^v_\kappa \boldsymbol{\Pi\, \alpha : \kappa.\, \kappa'}}\ \text{CPS}^v_\kappa\text{-PiK}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{K}' \leadsto^v_A \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{K} : \mathsf{U} \leadsto^v_\kappa \boldsymbol{\kappa}}{\Gamma \vdash \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{K} : \mathsf{U} \leadsto^v_\kappa \boldsymbol{\Pi\, \mathbf{x} : \mathbf{A}.\, \kappa}}\ \text{CPS}^v_\kappa\text{-PiA}$$

Figure I.3: CPS^v of Kinds

$$\boxed{\Gamma \vdash A : K \rightsquigarrow^v_A \mathbf{A} \text{ Lemma 6.6.6 will show } \Gamma^+ \vdash A^+ : K^+}$$

$$\frac{}{\Gamma \vdash \alpha : K \rightsquigarrow^v_A \boldsymbol{\alpha}}\ \text{CPS}^v_A\text{-Var} \qquad \frac{\Gamma \vdash A : K' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : K \rightsquigarrow^v_{A\div} \mathbf{B}}{\Gamma \vdash \Pi x : A.\, B : K \rightsquigarrow^v_A \mathbf{\Pi\, x : A.\, B}}\ \text{CPS}^v_A\text{-Pi}$$

$$\frac{\Gamma \vdash K : U' \rightsquigarrow^v_\kappa \boldsymbol{\kappa} \qquad \Gamma, x : A \vdash B : U \rightsquigarrow^v_{A\div} \mathbf{B}}{\Gamma \vdash \Pi \alpha : K.\, B : U \rightsquigarrow^v_A \boldsymbol{\Pi\, \alpha : \kappa.\, \mathbf{B}}}\ \text{CPS}^v_A\text{-PiK}$$

$$\frac{\Gamma \vdash A : K' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : K \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \lambda x : A.\, B : \Pi x : A.\, K \rightsquigarrow^v_A \boldsymbol{\lambda}\, \mathbf{x : A.\, B}}\ \text{CPS}^v_A\text{-Constr}$$

$$\frac{\Gamma \vdash K : U \rightsquigarrow^v_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : K \vdash B : K' \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \lambda \alpha : K.\, B : \Pi \alpha : K.\, K' \rightsquigarrow^v_A \boldsymbol{\lambda\, \alpha : \kappa.}\, \mathbf{B}}\ \text{CPS}^v_A\text{-Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \Pi x : B.\, K \rightsquigarrow^v_A \mathbf{A} \\ \Gamma, x : A \vdash B : K' \rightsquigarrow^v_A \mathbf{B} \qquad \Gamma \vdash e : B \rightsquigarrow^v_e \mathbf{e}\end{array}}{\Gamma \vdash A\ e : K[e/x] \rightsquigarrow^v_A \mathbf{A}\ (\mathbf{e\ B\ id})}\ \text{CPS}^v_A\text{-AppConstr}$$

$$\frac{\Gamma \vdash A : \Pi \alpha : K'.\, K \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma \vdash B : K' \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash A\ B : K[B/\alpha] \rightsquigarrow^v_A \mathbf{A\ B}}\ \text{CPS}^v_A\text{-Inst}$$

$$\frac{\Gamma \vdash A : \star \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \Sigma x : A.\, B : \star \rightsquigarrow^v_A \mathbf{\Sigma\, x : A.\, B}}\ \text{CPS}^v_A\text{-Sigma}$$

$$\frac{}{\Gamma \vdash \mathsf{bool} : \star \rightsquigarrow^v_A \mathbf{bool}}\ \text{CPS}^v_A\text{-Bool}$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow^v_e \mathbf{e} \qquad \Gamma \vdash A : K' \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, x = e : A \vdash B : K \rightsquigarrow^v_A \mathbf{B}}{\Gamma \vdash \mathsf{let}\, x = e : A \,\mathsf{in}\, B : K \rightsquigarrow^v_A \mathbf{let\ x = e\ \ A\ \ id : A\ in\ B}}\ \text{CPS}^v_A\text{-Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : K \rightsquigarrow^v_A \mathbf{A} \\ \vdash \Gamma \rightsquigarrow^v \mathbf{\Gamma} \qquad \mathbf{\Gamma \vdash A} : \boldsymbol{\kappa'} \qquad \Gamma, \alpha = A : K \vdash B : \_ \rightsquigarrow^v_e \mathbf{B}\end{array}}{\Gamma \vdash \mathsf{let}\, \alpha = A : K \,\mathsf{in}\, B : \_ \rightsquigarrow^v_A \mathbf{let}\ \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa'}\ \mathbf{in\ B}}\ \text{CPS}^v_A\text{-LetK}$$

$$\frac{\Gamma \vdash A : K' \qquad \Gamma \vdash K \equiv K' \qquad \Gamma \vdash A : K' \rightsquigarrow^n_A \mathbf{A}}{\Gamma \vdash A : K \rightsquigarrow^n_A \mathbf{A}}\ \text{CPS}^v_A\text{-Conv}$$

$$\boxed{\Gamma \vdash A : \star \rightsquigarrow^v_{A\div} \mathbf{A} \text{ Lemma 6.6.6 will show } \Gamma^+ \vdash A^\div : \star^+}$$

$$\frac{\Gamma \vdash A : \star \rightsquigarrow^v_A \mathbf{A}}{\Gamma \vdash A : \star \rightsquigarrow^v_{A\div} \boldsymbol{\Pi\, \alpha : \star.\, (\mathbf{A} \to \alpha) \to \alpha}}\ \text{CPS}^v_A\text{-Comp}$$

Figure I.4: CPS^v of Types

$\boxed{\Gamma \vdash e : A \leadsto^v_e \mathbf{e} \text{ Lemma 6.6.6 will show } \Gamma^+ \vdash e^{\div} : A^{\div}}$

$$\frac{\Gamma \vdash A : K \leadsto^v_A \mathbf{A}}{\Gamma \vdash x : A \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : A \to \alpha.\, k\ x}}\ \text{CPS}^v_e\text{-}\textsc{Var}$$

$$\frac{\Gamma \vdash A : K' \leadsto^v_A \mathbf{A} \qquad \Gamma, x : A \vdash B : K \leadsto^v_{A\div} \mathbf{B} \qquad \Gamma, x : A \vdash e : B \leadsto^v_e \mathbf{e}}{\Gamma \vdash \lambda\, x : A.\, e : \Pi\, x : A.\, B \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : (\Pi\, x : A.\, B) \to \alpha.\, k\ (\lambda\, x : A.\, e)}}\ \text{CPS}^v_e\text{-}\textsc{Fun}$$

$$\frac{\Gamma \vdash K : \_ \leadsto^v_\kappa \boldsymbol{\kappa} \qquad \Gamma, \alpha : K \vdash B : \_ \leadsto^v_{A\div} \mathbf{B} \qquad \Gamma, \alpha : K \vdash e : B \leadsto^v_e \mathbf{e}}{\begin{array}{l}\Gamma \vdash \lambda\, \alpha : K.\, e : \Pi\, \alpha : K.\, B \leadsto^v_e \boldsymbol{\lambda\, \alpha_{ans} : \star.\, \lambda\, k : (\Pi\, \alpha : \kappa.\, B) \to \alpha_{ans}.} \\ \qquad\qquad\qquad\qquad\qquad\qquad \boldsymbol{k\ (\lambda\, \alpha : \kappa.\, e)}\end{array}}\ \text{CPS}^v_e\text{-}\textsc{Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \Pi\, x : A.\, B \leadsto^v_e \mathbf{e} \qquad \Gamma, x : A \vdash B : K \leadsto^v_{A\div} \mathbf{B}^{\div} \\ \Gamma, x : A \vdash B : K \leadsto^v_A \mathbf{B}^{+} \qquad \Gamma \vdash e' : A \leadsto^v_e \mathbf{e}' \qquad \Gamma \vdash A : K' \leadsto^v_A \mathbf{A}\end{array}}{\begin{array}{l}\Gamma \vdash e\ e' : B[e'/x] \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : (B^{+}[(e'\ A\ id)/x]) \to \alpha.} \\ \qquad\qquad\qquad\qquad \boldsymbol{e\ \alpha\ (\lambda\, f : \Pi\, x : A.\, B^{\div}.} \\ \qquad\qquad\qquad\qquad\qquad \boldsymbol{e'\ @\ \alpha\ (\lambda\, x : A.\, (f\ x)\ \alpha\ k))}\end{array}}\ \text{CPS}^v_e\text{-}\textsc{App}$$

$$\frac{\Gamma \vdash e : \Pi\, \alpha : K.\, B \leadsto^v_e \mathbf{e} \qquad \Gamma, \alpha : K \vdash B : \_ \leadsto^v_{A\div} \mathbf{B} \qquad \Gamma \vdash A : K \leadsto^v_e \mathbf{A}}{\begin{array}{l}\Gamma \vdash e\ A : \mathsf{let}\, x = A\, \mathsf{in}\, B \leadsto^v_e \boldsymbol{\lambda\, \alpha_{ans} : \star.\, \lambda\, k : (B[A/\alpha]) \to \alpha_{ans}.} \\ \qquad\qquad\qquad\qquad \boldsymbol{e\ \alpha\ (\lambda\, f : \Pi\, \alpha : \kappa.\, B.} \\ \qquad\qquad\qquad\qquad\quad \boldsymbol{(f\ A)\ \alpha_{ans}\ k)}\end{array}}\ \text{CPS}^v_e\text{-}\textsc{Inst}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : bool \to \alpha.\, k\ true}}\ \text{CPS}^n_e\text{-}\textsc{True}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \leadsto^n_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : bool \to \alpha.\, k\ false}}\ \text{CPS}^n_e\text{-}\textsc{False}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{bool} \leadsto^v_e \mathbf{e} \\ \Gamma \vdash B : \star \leadsto^v_A \mathbf{B} \qquad \Gamma \vdash e_1 : B \leadsto^v_e \mathbf{e}_1 \qquad \Gamma \vdash e_2 : B \leadsto^v_e \mathbf{e}_2\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : B \to \alpha.} \\ \qquad\qquad\qquad \boldsymbol{e\ @\ \alpha\ (\lambda\, x : bool.\, if\, x\, then\, (e_1\ @\ \alpha\ k)} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \boldsymbol{else\, (e_2\ @\ \alpha\ k))}\end{array}}\ \text{CPS}^n_e\text{-}\textsc{If}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : A \leadsto^v_e \mathbf{e} \\ \Gamma \vdash A : K' \leadsto^v_A \mathbf{A} \qquad \Gamma \vdash B : K \leadsto^v_A \mathbf{B} \qquad \Gamma, x = e : A \vdash e' : B \leadsto^v_e \mathbf{e}'\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\, x = e : A\, \mathsf{in}\, e' : B[e/x] \leadsto^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : B[(e\ A\ id)/x] \to \alpha.} \\ \qquad\qquad\qquad\qquad\qquad \boldsymbol{e\ @\ \alpha\ (\lambda\, x : A.\, e'\ \alpha\ k)}\end{array}}\ \text{CPS}^v_e\text{-}\textsc{Let}$$

Figure I.5: CPS^v of Terms (1/2)

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{A} \rightsquigarrow^v_e \mathbf{e}_1 \\ \Gamma \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}] \rightsquigarrow^v_e \mathbf{e}_2 \qquad \Gamma \vdash \mathsf{A} : \star \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^v_A \mathbf{B}\end{array}}{\begin{array}{l}\Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : \Sigma\, x : A.\, B \to \alpha}. \\ \qquad \mathbf{e}_1 \ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x_1 : A}. \\ \qquad\qquad \mathbf{e}_2 \ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x_2 : B}[(\mathbf{e}_1\ \mathbf{A}\ \mathbf{id})/\mathbf{x}]. \\ \qquad\qquad\qquad \mathbf{k}\ \langle \mathbf{x}_1, \mathbf{x}_2 \rangle\ \mathbf{as}\ \boldsymbol{\Sigma\, x : A.\, B}))\end{array}}\ \text{CPS}^v_e\text{-Pair}$$

$$\frac{\Gamma \vdash \mathsf{A} : \star \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^v_e \mathbf{e}}{\begin{array}{l}\Gamma \vdash \mathsf{fst}\, \mathsf{e} : \mathsf{A} \rightsquigarrow^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : A}^{+} \boldsymbol{\to \alpha}. \\ \qquad \mathbf{e}\ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, y : \Sigma\, x : A.\, B.}\ \mathbf{let}\ \mathbf{z} = \mathbf{fst}\ \mathbf{y}\ \mathbf{in}\ \mathbf{k}\ \mathbf{z})\end{array}}\ \text{CPS}^v_e\text{-Fst}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \star \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^v_A \mathbf{B} \\ \Gamma \vdash (\mathsf{fst}\, \mathsf{e}) : \mathsf{A} \rightsquigarrow^v_e (\mathsf{fst}\, \mathsf{e})^{\div} \qquad \Gamma \vdash \mathsf{e} : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^v_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{snd}\, \mathsf{e} : \mathsf{B}[\mathsf{fst}\, \mathsf{e}/\mathsf{x}] \rightsquigarrow^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : B}[((\mathsf{fst}\, \mathsf{e})^{\div}\ \mathbf{A}\ \mathbf{id})/\mathbf{x}] \boldsymbol{\to \alpha}. \\ \qquad \mathbf{e}\ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, y : \Sigma\, x : A.\, B.}\ \mathbf{let}\ \mathbf{z} = \mathbf{snd}\ \mathbf{y}\ \mathbf{in}\ \mathbf{k}\ \mathbf{z})\end{array}}\ \text{CPS}^v_e\text{-Snd}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{A} \rightsquigarrow^v_e \mathbf{e}_1 \\ \Gamma \vdash \mathsf{e}_2 : \mathsf{B}[\mathsf{e}_1/\mathsf{x}] \rightsquigarrow^v_e \mathbf{e}_2 \qquad \Gamma \vdash \mathsf{A} : \star \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \star \rightsquigarrow^v_A \mathbf{B}\end{array}}{\begin{array}{l}\Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \Sigma\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} \rightsquigarrow^v_e \boldsymbol{\lambda\, \alpha : \star.\, \lambda\, k : \Sigma\, x : A.\, B \to \alpha}. \\ \qquad \mathbf{e}_1 \ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x : A}. \\ \qquad\qquad \mathbf{e}_2 \ \boldsymbol{@}\ \boldsymbol{\alpha}\ (\boldsymbol{\lambda\, x_2 : B.}\ \mathbf{k}\ \langle \mathbf{x}, \mathbf{x}_2 \rangle))\end{array}}\ \text{CPS}^v_e\text{-Pair-Alt}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^v_\kappa \boldsymbol{\kappa} \\ \Gamma, \alpha = \mathsf{A} : \mathsf{K} \vdash \mathsf{B} : \mathsf{K}' \rightsquigarrow^v_A \mathbf{B} \qquad \Gamma, \alpha = \mathsf{A} : \mathsf{K} \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^v_e \mathbf{e}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\, \alpha = \mathsf{A} : \mathsf{K}\, \mathsf{in}\, \mathsf{e} : \mathsf{B}[\mathsf{A}/\alpha] \rightsquigarrow^v_e \boldsymbol{\lambda\, \alpha_{ans} : \star.\, \lambda\, k : B}[\mathbf{A}/\boldsymbol{\alpha}] \boldsymbol{\to \alpha_{ans}}. \\ \qquad \mathbf{let}\ \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}\ \mathbf{in}\ \mathbf{e}\ \boldsymbol{\alpha_{ans}}\ \mathbf{k}\end{array}}\ \text{CPS}^v_e\text{-LetK}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow^v_e \mathbf{e}}{\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^v_e \mathbf{e}}\ \text{CPS}^v_e\text{-Conv}$$

Figure I.6: CPS^v of Terms (2/2)

$\boxed{\vdash \Gamma \rightsquigarrow^v \mathbf{\Gamma} \text{ Lemma 6.6.6 will show } \vdash \Gamma^+}$

$$\frac{}{\vdash \cdot \rightsquigarrow^v \cdot}\ \text{CPS}^v_\Gamma\text{-Empty} \qquad \frac{\vdash \Gamma \rightsquigarrow^v \mathbf{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^v_A \mathbf{A}}{\vdash \Gamma, \mathsf{x} : \mathsf{A} \rightsquigarrow^v \mathbf{\Gamma}, \mathbf{x} : \mathbf{A}}\ \text{CPS}^v_\Gamma\text{-AssumT}$$

$$\frac{\vdash \Gamma \rightsquigarrow^v \mathbf{\Gamma} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^v_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha : \mathsf{K} \rightsquigarrow^v \mathbf{\Gamma}, \boldsymbol{\alpha} : \boldsymbol{\kappa}}\ \text{CPS}^v_\Gamma\text{-AssumK}$$

$$\frac{\vdash \Gamma \rightsquigarrow^v \mathbf{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow^v_e \mathbf{e}}{\vdash \Gamma, \mathsf{x} = \mathsf{e} : \mathsf{A} \rightsquigarrow^v \mathbf{\Gamma}, \mathbf{x} = \mathbf{e}\ \mathbf{A}\ \mathbf{id} : \mathbf{A}}\ \text{CPS}^v_\Gamma\text{-Def}$$

$$\frac{\vdash \Gamma \rightsquigarrow^v \mathbf{\Gamma} \qquad \Gamma \vdash \mathsf{A} : \mathsf{K} \rightsquigarrow^v_A \mathbf{A} \qquad \Gamma \vdash \mathsf{K} : \mathsf{U} \rightsquigarrow^v_\kappa \boldsymbol{\kappa}}{\vdash \Gamma, \alpha = \mathsf{A} : \mathsf{K} \rightsquigarrow^v \mathbf{\Gamma}, \boldsymbol{\alpha} = \mathbf{A} : \boldsymbol{\kappa}}\ \text{CPS}^v_\Gamma\text{-DefT}$$

Figure I.7: CPS^v of Environments

J REFERENCE FOR COC^{CC}

This appendix contains the complete definitions for CoC^{CC}. All CoC^{CC} figures from Chapter 7 are reproduced and completed with elided parts here, and elided figures are presented here. These figures contained extensions with definitions required to fully formalize the compiler correctness results, including booleans with non-dependent elimination.

Typographical Note. *In this appendix, I typeset CoC^{CC} in a* **bold, red, serif font**.

$$\begin{array}{llll}
\textit{Universes} & \mathbf{U} & ::= & \star \mid \square \\
\textit{Expressions} & \mathbf{e}, \mathbf{A}, \mathbf{B} & ::= & \mathbf{x} \mid \star \mid \mathbf{Code}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{B} \\
 & & \mid & \mathbf{e} \Rightarrow \mathbf{Code}\,\mathbf{n} : \mathbf{A}'.\,\mathbf{B} \mid \boldsymbol{\lambda}\,(\mathbf{n} : \mathbf{A}', \mathbf{x} : \mathbf{A}).\,\mathbf{e} \mid \mathbf{e}_f\ \mathbf{e}_1\ \mathbf{e}_2 \\
 & & \mid & \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \mid \langle \mathbf{e}, \mathbf{e} \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\,\mathbf{x} : \mathbf{A}.\,\mathbf{B} \mid \mathbf{fst}\,\mathbf{e} \mid \mathbf{snd}\,\mathbf{e} \\
 & & \mid & \mathbf{bool} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\,\mathbf{e}\,\mathbf{then}\,\mathbf{e}_1\,\mathbf{else}\,\mathbf{e}_2 \\
 & & \mid & \mathbf{let}\,\mathbf{x} = \mathbf{e} : \mathbf{A}\,\mathbf{in}\,\mathbf{e} \mid \exists\,(\boldsymbol{\alpha} : \mathbf{A}, \mathbf{n} : \mathbf{A}').\,\mathbf{B} \\
 & & \mid & \mathbf{pack}\,\langle \mathbf{A}, \mathbf{e}', \mathbf{e} \rangle\ \mathbf{as}\,\exists\,(\boldsymbol{\alpha} : \mathbf{A}, \mathbf{n} : \mathbf{A}').\,\mathbf{B} \\
 & & \mid & \mathbf{unpack}\,\langle \mathbf{x}, \mathbf{x}, \mathbf{x} \rangle = \mathbf{e}\,\mathbf{in}\,\mathbf{e} \\
\textit{Environments} & \boldsymbol{\Gamma} & ::= & \cdot \mid \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A} \mid \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}
\end{array}$$

Figure J.1: CoC^{CC} Syntax

$$\begin{array}{rcl}
\mathbf{e} : \mathbf{t} & \stackrel{\text{def}}{=} & \boldsymbol{\Gamma} \vdash \mathbf{e} : \mathbf{A} \text{ where } \boldsymbol{\Gamma} \text{ is obvious from context} \\
\mathbf{pack}\,\langle \mathbf{e}_i \ldots \rangle & \stackrel{\text{def}}{=} & \mathbf{pack}\,\langle \mathbf{e}_i \ldots \rangle\ \mathbf{as}\,\exists\,\mathbf{x}_i : \mathbf{t}_i \ldots.\ \text{ where } \mathbf{e}_i : \mathbf{t}_i[\mathbf{e}_{i-1}/\mathbf{x}_{i-1}] \\
\langle \mathbf{e}_i \ldots \rangle & \stackrel{\text{def}}{=} & \langle \mathbf{e}_i \ldots \rangle\ \mathbf{as}\,\boldsymbol{\Sigma}\,(\mathbf{x}_i : \mathbf{t}_i \ldots) \text{ where } \mathbf{e}_i : \mathbf{t}_i[\mathbf{e}_{i-1}/\mathbf{x}_{i-1}] \\
\mathbf{let}\,\mathbf{x} = \mathbf{e}\,\mathbf{in}\,\mathbf{e}' & \stackrel{\text{def}}{=} & \mathbf{let}\,\mathbf{x} = \mathbf{e} : \mathbf{t}\,\mathbf{in}\,\mathbf{e}' \text{ where } \mathbf{e} : \mathbf{t} \\
\mathbf{let}\,\langle \mathbf{x}_i \ldots \rangle = \mathbf{e}\,\mathbf{in}\,\mathbf{e}' & \stackrel{\text{def}}{=} & \mathbf{let}\,\mathbf{x}_0 = \boldsymbol{\pi}_0\,\mathbf{e}\,\mathbf{in}\ \ldots \mathbf{let}\,\mathbf{x}_{|i|} = \boldsymbol{\pi}_{|i|}\ \mathbf{in}\,\mathbf{e}' \\
\mathbf{e}[\langle \mathbf{x}_i \ldots \rangle / \mathbf{e}] & \stackrel{\text{def}}{=} & \mathbf{e}[\mathbf{x}_i / \boldsymbol{\pi}_i\,\mathbf{e}]
\end{array}$$

Figure J.2: CoC^{CC} Syntactic Sugar

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

$$\begin{array}{rll}
\Gamma \vdash (\lambda\, x' : A', x : A.\, e_1)\; e'\; e & \rhd_\beta & e_1[e'/x'][e/x] \\
\Gamma \vdash \mathrm{fst}\, \langle e_1, e_2 \rangle & \rhd_{\pi_1} & e_1 \\
\Gamma \vdash \mathrm{snd}\, \langle e_1, e_2 \rangle & \rhd_{\pi_2} & e_2 \\
\Gamma \vdash \mathrm{if\ true\ then}\, e_1\, \mathrm{else}\, e_2 & \rhd_{\iota_1} & e_1 \\
\Gamma \vdash \mathrm{if\ false\ then}\, e_1\, \mathrm{else}\, e_2 & \rhd_{\iota_2} & e_2 \\
\Gamma \vdash x & \rhd_\delta & e \\
 & & \text{where } x = e : A \in \Gamma \\
\Gamma \vdash \mathrm{let}\, x = e : A\, \mathrm{in}\, e_1 & \rhd_\zeta & e_1[e/x] \\
\Gamma \vdash \mathrm{unpack}\, \langle x, x', x_b \rangle = \mathrm{pack}\, \langle e, e', e_b \rangle\, \mathrm{in}\, e_2 & \rhd_\exists & e_2[\langle e, e', e_b \rangle / \langle x, x', x_b \rangle]
\end{array}$$

Figure J.3: CoC^{CC} Reduction

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

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \rhd^* A_1' \\ \Gamma, n : A_1' \vdash A_2 \rhd^* A_2' \qquad \Gamma, n : A_1', x : A_2' \vdash B \rhd^* B'\end{array}}{\Gamma \vdash \mathrm{Code}\, n : A_1, x : A_2.\, B \rhd^* \mathrm{Code}\, n : A_1', x : A_2'.\, B'}\ \textsc{Red-Cong-T-Code}$$

$$\frac{\Gamma \vdash e \rhd^* e' \qquad \Gamma \vdash A_2 \rhd^* A_2' \qquad \Gamma, x : A_2' \vdash B \rhd^* B'}{\Gamma \vdash e \Rightarrow \mathrm{Code}\, x : A_2.\, B \rhd^* e' \Rightarrow \mathrm{Code}\, x : A_2'.\, B'}\ \textsc{Red-Cong-}\Rightarrow$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \rhd^* A_1' \\ \Gamma, n : A_1' \vdash A_2 \rhd^* A_2' \qquad \Gamma, n : A_1', x : A_2' \vdash e \rhd^* e'\end{array}}{\Gamma \vdash \lambda\, (n : A_1, x : A_2).\, e \rhd^* \lambda\, (n : A_1', x : A_2').\, e'}\ \textsc{Red-Cong-Code}$$

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

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

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

$$\frac{\Gamma \vdash e \rhd^* e'}{\Gamma \vdash \mathrm{fst}\, e \rhd^* \mathrm{fst}\, e'}\ \textsc{Red-Cong-Fst} \qquad \frac{\Gamma \vdash e \rhd^* e'}{\Gamma \vdash \mathrm{snd}\, e \rhd^* \mathrm{snd}\, e'}\ \textsc{Red-Cong-Snd}$$

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

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash A \rhd^* A' \qquad \Gamma, x = e' : A' \vdash e_2 \rhd^* e_2'}{\Gamma \vdash \mathrm{let}\, x = e_1 : A\, \mathrm{in}\, e_2 \rhd^* \mathrm{let}\, x = e_1' : A'\, \mathrm{in}\, e_2'}\ \textsc{Red-Cong-Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \rhd^* A_1' \\ \Gamma, x : A_1' \vdash A_2 \rhd^* A_2' \qquad \Gamma, x : A_1', x' : A_2' \vdash B \rhd^* B'\end{array}}{\Gamma \vdash \exists\, (x : A, x' : A').\, B \rhd^* \exists\, (x : A_1, x' : A_2).\, B'}\ \textsc{Red-Cong-Exist}$$

$$\frac{\Gamma \vdash A \rhd^* A' \qquad \Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma \vdash e_2 \rhd^* e_2' \qquad \Gamma \vdash e_3 \rhd^* e_3'}{\Gamma \vdash \mathrm{pack}\, \langle e_1, e_2, e_3 \rangle\ \mathrm{as}\ A \rhd^* \mathrm{pack}\, \langle e_1', e_2', e_3' \rangle\ \mathrm{as}\ A'}\ \textsc{Red-Cong-Pack}$$

$$\frac{\Gamma \vdash e_1 \rhd^* e_1' \qquad \Gamma, x : A, x' : A', x_b : B \vdash e_2 \rhd^* e_2}{\Gamma \vdash \mathrm{unpack}\, \langle x, x', x_b \rangle = e_1\, \mathrm{in}\, e_2 \rhd^* \mathrm{unpack}\, \langle x, x', x_b \rangle = e_1'\, \mathrm{in}\, e_2'}\ \textsc{Red-Cong-Unpack}$$

Figure J.4: CoC^{CC} Conversion

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \triangleright^* \mathsf{pack}\,\langle A', e', \lambda\,(n : A', x : A).\, e_1' \rangle \\ \Gamma \vdash e_2 \triangleright^* e_2' \qquad \Gamma, x : A[e'/n] \vdash e_1'[e'/n] \equiv \mathsf{unpack}\,\langle \alpha, n, f\rangle = e_2'\ \mathsf{in}\, f\ n\ x\end{array}}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-}\eta_1$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \triangleright^* e_1' \qquad \Gamma \vdash e_2 \triangleright^* \mathsf{pack}\,\langle A', e', \lambda\,(n : A', x : A).\, e_2' \rangle \\ \Gamma, x : A[e'/n] \vdash \mathsf{unpack}\,\langle \alpha, n, f\rangle = e_1'\ \mathsf{in}\, f\ n\ x \equiv e_2'[e'/n]\end{array}}{\Gamma \vdash e_1 \equiv e_2}\ \equiv\text{-}\eta_2$$

Figure J.5: CoC^{CC} Equivalence

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

$$\frac{(x : A \in \Gamma \text{ or } x = e : A \in \Gamma) \qquad \vdash \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad\qquad \frac{\vdash \Gamma}{\Gamma \vdash \star : \square}\ *$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma\, x : A.\, B : \star}\ \textsc{Sig} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B : \Sigma\, x : A.\, B}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{fst}\, e : A}\ \textsc{Fst} \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B}{\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x]}\ \textsc{Snd} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{bool} : \star}\ \textsc{Bool}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \textsc{True} \qquad\qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : B \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B}\ \textsc{If}$$

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

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash B : U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e : B}\ \textsc{Conv}$$

Figure J.6: CoC^{CC} Typing (1/2)

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

$$\frac{\Gamma \vdash B : \star}{\Gamma \vdash \mathrm{Code}\,(x' : A', x : A).\, B : \star}\ \textsc{Code-*}$$

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

$$\frac{\Gamma \vdash e : \mathrm{Code}\,(x' : A', x : A).\, B \qquad \Gamma \vdash e' : A'}{\Gamma \vdash e : e' \Rightarrow \mathrm{Code}\,(x : A[e'/x']).\, B[e'/x']}\ \textsc{TrFun}$$

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

$$\frac{\Gamma \vdash f : e' \Rightarrow \mathrm{Code}\, x : A.\, B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e'\ e : B[e/x]}\ \textsc{TrApp}$$

$$\frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash A' : U' \qquad \Gamma, x : A, x' : A' \vdash B : \star}{\Gamma \vdash \exists\,(x : A, x' : A').\, B : \star}\ \textsc{Exist}$$

$$\frac{\begin{array}{c}\Gamma \vdash \exists\,(x : A, x' : A').\, B : U \\ \Gamma \vdash e : A \qquad \Gamma \vdash e' : A'[e/x] \qquad \Gamma \vdash e_b : B[e/x][e'/x']\end{array}}{\Gamma \vdash \mathrm{pack}\,\langle e, e', e_b\rangle\ \mathrm{as}\ \exists\,(x : A, x' : A').\, B : \exists\,(x : A, x' : A').\, B}\ \textsc{Pack}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \exists\,(x : A, x' : A').\, B \\ \Gamma, x : A, x' : A', x_b : B \vdash e' : B' \qquad \Gamma \vdash B' : U\end{array}}{\Gamma \vdash \mathrm{unpack}\,\langle x, x', x_b\rangle = e\ \mathrm{in}\ e' : B'}\ \textsc{Unpack}$$

Figure J.7: CoC^{CC} Typing (2/2)

$\boxed{\vdash \Gamma}$

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

$$\frac{\vdash \Gamma \qquad \Gamma \vdash e : A \qquad \Gamma \vdash A : U}{\vdash \Gamma, x = e : A}\ \textsc{W-Def}$$

Figure J.8: CoC^{CC} Well-formed Environments

CoC^{CC} Observations $\quad v \quad ::= \quad \mathbf{true} \mid \mathbf{false}$

Figure J.9: CoC^{CC} Observations

$\boxed{\Gamma \vdash \mathbf{e}}$ $\qquad\qquad$ $\boxed{\vdash \mathbf{e}}$

$$\frac{\Gamma \vdash \mathbf{e} : \mathbf{bool}}{\Gamma \vdash \mathbf{e}} \qquad\qquad \frac{\cdot \vdash \mathbf{e}}{\vdash \mathbf{e}}$$

Figure J.10: CoC^{CC} Components and Programs

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

$$\mathbf{eval}(\mathbf{e}) \quad = \quad \mathbf{v} \quad \text{if} \vdash \mathbf{e} \text{ and } \cdot \vdash \mathbf{e} \triangleright^* \mathbf{v}$$

Figure J.11: CoC^{CC} Evaluation

$$\textit{Closing Substitutions} \quad \gamma \quad \stackrel{\text{def}}{=} \quad \cdot \mid \gamma[\mathbf{x} \mapsto \mathbf{e}]$$

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

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

$\boxed{\gamma(\mathbf{e}) = \mathbf{e}}$

$$\cdot(\mathbf{e}) = \mathbf{e} \qquad\qquad \gamma[\mathbf{x} \mapsto \mathbf{e}'](\mathbf{e}) = \gamma(\mathbf{e}[\mathbf{x}/\mathbf{e}'])$$

Figure J.12: CoC^{CC} Closing Substitutions and Linking

```
Definition translucent {A} e B := forall (x : A), (x = e) -> B.

Notation "e ⇒ B" := (translucent e B) (at level 42).

Definition translucent_intro {A B} {e' : A}
  (e : forall (x : A), B x) : e' ⇒ B e'
:= fun x w => match w with eq_refl => e x end.

Notation "e :T: A" := (translucent_intro e : A) (at level 99).

Definition translucent_elim {A B} {e' : A} (e : e' ⇒ B) : B
:= e e' eq_refl.

Notation "e @ m" := (@translucent_elim _ _ m e) (at level 99).

Inductive sigS {A:Type} (P:A -> Type) : Prop :=
  existS : forall x:A, P x -> sigS P.

Notation "'∃' x : A , P" := (sigS (A:=A) (fun x => P))
  (at level 0, x at level 99) : type_scope.

Definition Closure A B : Prop :=
  ∃ α : Type , ∃ env : α , (env ⇒ (A -> B)).
```

Listing J.1: Model of CoC^{CC} in Coq

K REFERENCE FOR PARAMETRIC CLOSURE CONVERSION

This appendix contains the complete definitions for the parametric closure conversion from CoC^{D} to CoC^{CC}. All translation figures from Chapter 7 are reproduced and completed with elided parts here, and elided figures are presented here.

Typographical Note. *In this appendix, I typeset the source language, CoC^{D}, in a* blue, non-bold, sans-serif font*, and the target language, CoC^{CC}, in a* **bold, red, serif font***.*

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

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

Figure K.1: Observation Relation between CoC^{D} and CoC^{CC}

$$\begin{array}{lcl} \mathrm{FV}(\mathsf{e}, \mathsf{B}, \mathsf{\Gamma}) & \overset{\text{def}}{=} & \mathsf{\Gamma}_0, \dots, \mathsf{\Gamma}_n, (\mathsf{x}_0 : \mathsf{A}_0, \dots, \mathsf{x}_n : \mathsf{A}_n) \\ & \textit{where} & \mathsf{x}_0, \dots, \mathsf{x}_n = \mathrm{fv}(\mathsf{e}, \mathsf{B}) \\ & & \mathsf{\Gamma} \vdash \mathsf{x}_0 : \mathsf{A}_0 \\ & & \quad\vdots \\ & & \mathsf{\Gamma} \vdash \mathsf{x}_n : \mathsf{A}_n \\ & & \mathsf{\Gamma}_0 = \mathrm{FV}(\mathsf{A}_0, \_, \mathsf{\Gamma}) \\ & & \qquad\vdots \\ & & \mathsf{\Gamma}_n = \mathrm{FV}(\mathsf{A}_n, \_, \mathsf{\Gamma}) \end{array}$$

Figure K.2: Dependent Free Variable Sequences

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

$$\frac{\Gamma \vdash A : U \rightsquigarrow \mathbf{A} \qquad \Gamma, x : A \vdash B : \star \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Pi\, x : A.\, B : U' \rightsquigarrow \boldsymbol{\exists}\, (\boldsymbol{\alpha} : \boldsymbol{\star}, \mathbf{n} : \boldsymbol{\alpha}).\, \mathbf{n} \Rightarrow \mathbf{Code}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\ \text{CC-Pi}$$

$$\frac{\begin{array}{c}\Gamma, x : t \vdash e : t' \rightsquigarrow \mathbf{e} \qquad \Gamma \vdash t : U \rightsquigarrow \mathbf{t} \qquad \Gamma, x : t \vdash t' : U \rightsquigarrow \mathbf{t'} \\ x_i : A_i \cdots = \mathrm{FV}(\lambda\, x : A.\, e, \Pi\, x : A.\, B, \Gamma) \qquad \Gamma \vdash A_i : U \rightsquigarrow \mathbf{A}_i \ldots \end{array}}{\begin{array}{l}\Gamma \vdash \lambda\, x : t.\, e : \Pi\, x : t.\, t' \rightsquigarrow \mathbf{pack}\, \langle \boldsymbol{\Sigma}\, (\mathbf{x}_i : \mathbf{A}_i \ldots), \langle \mathbf{x}_i \ldots \rangle, \\ \qquad\qquad\qquad \boldsymbol{\lambda}\, (\mathbf{n} : \boldsymbol{\Sigma}\, (\mathbf{x}_i : \mathbf{A}_i \ldots), \qquad). \\ \qquad\qquad\qquad\quad \mathbf{x} : \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\, \mathbf{in}\, \mathbf{t} \\ \qquad\qquad\qquad\quad \mathbf{let}\, \langle \mathbf{x}_i \ldots \rangle = \mathbf{n}\, \mathbf{in}\, \mathbf{e} \rangle \end{array}}\ \text{CC-Lam}$$

$$\frac{\Gamma \vdash e_1 : \_ \rightsquigarrow \mathbf{e}_1 \qquad \Gamma \vdash e_2 : \_ \rightsquigarrow \mathbf{e}_2}{\Gamma \vdash e_1\ e_2 : t \rightsquigarrow \mathbf{unpack}\, \langle \boldsymbol{\alpha}, \mathbf{n}, \mathbf{f} \rangle = \mathbf{e}_1\, \mathbf{in}\, \mathbf{f}\ \ \mathbf{n}\ \ \mathbf{e}_2}\ \text{CC-App}$$

Figure K.3: Parametric Closure Conversion from CoC^{D} to CoC^{CC} (1/2)

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

$$\frac{}{\Gamma \vdash x : A \rightsquigarrow \mathbf{x}}\;\textsc{CC-Var} \qquad \frac{}{\Gamma \vdash \star : \square \rightsquigarrow \boldsymbol{\star}}\;\textsc{CC-*}$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \rightsquigarrow \mathbf{A} \qquad \Gamma, x : A \vdash B : \mathsf{Type}_i \rightsquigarrow \mathbf{B}}{\Gamma \vdash \Sigma\, x : A.\, B : \mathsf{Type}_i \rightsquigarrow \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\;\textsc{CC-Sig}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \rightsquigarrow \mathbf{e_1} \\ \Gamma \vdash e_2 : B[e_1/x] \rightsquigarrow \mathbf{e_2} \qquad \Gamma \vdash A : \mathsf{Type}_i \rightsquigarrow \mathbf{A} \qquad \Gamma, x : A \vdash B : \mathsf{Type}_i \rightsquigarrow \mathbf{B}\end{array}}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathsf{as}\ \Sigma\, x : A.\, B : \Sigma\, x : A.\, B \rightsquigarrow \langle \mathbf{e_1}, \mathbf{e_2} \rangle\ \mathbf{as}\ \boldsymbol{\Sigma}\, \mathbf{x} : \mathbf{A}.\, \mathbf{B}}\;\textsc{CC-Pair}$$

$$\frac{\Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow \mathbf{e}}{\Gamma \vdash \mathsf{fst}\, e : A \rightsquigarrow \mathbf{fst}\, \mathbf{e}}\;\textsc{CC-Fst} \qquad \frac{\Gamma \vdash e : \Sigma\, x : A.\, B \rightsquigarrow \mathbf{e}}{\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x] \rightsquigarrow \mathbf{snd}\, \mathbf{e}}\;\textsc{CC-Snd}$$

$$\frac{}{\Gamma \vdash \mathsf{bool} : \star \rightsquigarrow \mathbf{bool}}\;\textsc{CC-Bool} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \rightsquigarrow \mathbf{true}}\;\textsc{CC-True}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \rightsquigarrow \mathbf{false}}\;\textsc{CC-False}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \rightsquigarrow \mathbf{e} \qquad \Gamma \vdash e_1 : B \rightsquigarrow \mathbf{e_1} \qquad \Gamma \vdash e_2 : B \rightsquigarrow \mathbf{e_2}}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : B \rightsquigarrow \mathbf{if}\, \mathbf{e}\, \mathbf{then}\, \mathbf{e_1}\, \mathbf{else}\, \mathbf{e_2}}\;\textsc{CC-If}$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow \mathbf{e} \qquad \Gamma, x : A \vdash e' : B \rightsquigarrow \mathbf{e'}}{\Gamma \vdash \mathsf{let}\, x = e\, \mathsf{in}\, e' : B[e/x] \rightsquigarrow \mathbf{let}\, \mathbf{x} = \mathbf{e}\, \mathbf{in}\, \mathbf{e'}}\;\textsc{CC-Let} \qquad \frac{\Gamma \vdash e : A \rightsquigarrow \mathbf{e}}{\Gamma \vdash e : B \rightsquigarrow \mathbf{e}}\;\textsc{CC-Conv}$$

$\boxed{\vdash \Gamma \rightsquigarrow \boldsymbol{\Gamma} \quad \text{where} \vdash \Gamma}$

$$\frac{}{\vdash \cdot \rightsquigarrow \boldsymbol{\cdot}}\;\textsc{CC-Empty} \qquad \frac{\vdash \Gamma \rightsquigarrow \boldsymbol{\Gamma} \qquad \Gamma \vdash A : U \rightsquigarrow \mathbf{A}}{\vdash \Gamma, x : A \rightsquigarrow \boldsymbol{\Gamma}, \mathbf{x} : \mathbf{A}}\;\textsc{CC-Assum}$$

$$\frac{\vdash \Gamma \rightsquigarrow \boldsymbol{\Gamma} \qquad \Gamma \vdash e : A \rightsquigarrow \mathbf{e} \qquad \Gamma \vdash A : U \rightsquigarrow \mathbf{A}}{\vdash \Gamma, x = e : A \rightsquigarrow \boldsymbol{\Gamma}, \mathbf{x} = \mathbf{e} : \mathbf{A}}\;\textsc{CC-Def}$$

Figure K.4: Parametric Closure Conversion from CoC^{D} to CoC^{CC} (2/2)