COMBINING PROOFS AND PROGRAMS

Chris Casinghino

A DISSERTATION
in
Computer and Information Science
Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
2014

Supervisor of Dissertation

____

Stephanie Weirich
Associate Professor of CIS

Graduate Group Chairperson

____

Lyle Ungar
Professor of CIS

Dissertation Committee
Jean Gallier (Professor of CIS)
Benjamin C. Pierce (Henry Salvatori Professor of CIS; Committee Chair)
Aaron Stump (Professor of Computer Science, University of Iowa)
Steve Zdancewic (Associate Professor of CIS)

For my father, who taught me how to write.

Acknowledgments

When I was visiting prospective graduate schools seven years ago, my primary goal was to find a place with a large group of researchers who were excited and motivated by the same things that excite me. On that basis I chose Penn, and I'm happy to report that I was not mistaken. I am extremely grateful to the many faculty, postdocs and fellow graduate students I have worked with along the way—this document could not exist without them.

Thanks must go first to my advisor, Stephanie Weirich, for her unending patience and helpfulness. The metatheory of dependently typed languages is an extremely complex field, where small changes to a system can have drastic and hard to predict consequences in proofs that span hundreds of pages (or thousands of lines of Coq code). Stephanie's generosity with her time and her ability to rapidly take in a new language and predict problems before I explained them (or even noticed them) have been invaluable.

My closest collaborator at Penn has certainly been Vilhelm Sjöberg, whose related thesis you will read more about later. We spent countless fun afternoons puzzling through the tricky details of proofs about (usually flawed) language designs. The act of theorem proving is so often solitary—it is truly a joy to have a colleague who is working on closely related systems and is willing to, at the drop of a hat, talk through a complex proof or just listen to crazy ideas.

The PCC$^\theta$ language is a direct result of Trellys, a multi-institution collaborative research project aimed at making dependently typed languages more practical. I confess that I did not have high expectations for hammering out the specifics of a language design in the context of multi-day research meetings with 15-20 collaborators. I am delighted that working with the other members of the Trellys project has proved me wrong and been such a great pleasure. Thanks go the faculty at other the other institutions, Aaron Stump and Tim Sheard, and to their graduate students and postdocs, especially to those I have worked with most closely: Nathan Collins, Harley Eades, and Garrin Kimmell.

The weekly meetings of the Penn PL Club have always given me something to look forward to. The faculty involved, Benjamin Pierce and Steve Zdancewic, have been nothing but helpful and have demonstrated a repeated commitment to the success of everyone in the group (especially via excellent feedback on drafts and talks). I have been extremely lucky to spend time here with a group of graduate students who are working on so many interesting projects and who are always a pleasure to talk to. Thanks especially to my Hac φ co-organizers Daniel Wagner and Brent Yorgey, to my (long gone) officemates Aaron Bohannon and Karl Mazurak, to my fellow-traveler Michael Greenberg, and to Peter-Michael Osera, for endless fun conversations.

This work was supported by the National Science Foundation (NSF grant 0910500).

ABSTRACT

COMBINING PROOFS AND PROGRAMS

Chris Casinghino

Stephanie Weirich

Dependently typed languages allow us to develop programs and write proofs quickly and without errors. The last decade has seen many success stories for verified programming with dependent types. Despite these successes, dependently typed languages are rarely used for day-to-day programming tasks. There are many reasons why these languages have not been more widely adopted. This thesis addresses two of them: First, existing dependently typed languages restrict recursion and require programmers to prove that every function terminates. Second, traditional representations of equality are inconvenient to work with because they require too much typing information and because their eliminations clutter terms.

This thesis introduces PCC$^\theta$, a new dependently typed core language that addresses these problems. To handle potentially non-terminating computations, PCC$^\theta$ is split into two fragments: a *programmatic* fragment with support for general recursion, and a *logical* fragment that is restricted for consistency. Crucially, while the logical fragment is consistent, it can reason about programs written in the inconsistent programmatic fragment. To make equality reasoning easier, PCC$^\theta$ includes a novel heterogeneous notion of equality whose eliminations are not marked in terms.

The metatheory of PCC$^\theta$ is studied in detail, including a complete proof of normalization and consistency for its logical fragment. The normalization proof required the development of a novel technique, *partially step-indexed logical relations*, which is motivated and explained. Additionally, to demonstrate that PCC$^\theta$ addresses the problems described above, we have extended it to a complete core language Theta, adding features like user-defined datatypes and an infinite hierarchy of universes. Several examples are carried out in Theta, and an implementation is available.

Contents

List of Figures

Chapter 1

Introduction

From the entrance to the exit,
Is longer than it looks from where we stand.
I want to say I'm sorry for stuff I haven't done yet.
Things will shortly get completely out of hand.

Old College Try
The Mountain Goats

Dependently typed languages allow us to develop programs and write proofs quickly and without errors. The last decade has seen many success stories for dependently type programming. For example, the CompCert project has produced a formally verified C compiler [38]. Researchers have created concurrency libraries that guarantee programs are deadlock free [10], and security libraries that guarantee access-control and information flow properties [44]. Dependent types allow us to write programs whose types could not be captured by more traditional languages, as in generic programming [61, 60, 63]. And they have been used to verify substantial mathematical results, like the four color theorem and the Feit-Thompson theorem [29, 30].

Despite these successes, dependently typed languages are rarely used for day-to-day programming tasks. In this thesis, we introduce PCC$^\theta$, a new dependently typed core language that includes novel features which address two of the major problems for practical programming with dependent types. First, PCC$^\theta$ allows programmers to write arbitrary recursive functions and reason about them without sacrificing the consistency of its type system. Second, PCC$^\theta$ includes a novel notion of equality that ignores type annotations and whose uses do not pollute terms.

The next two sections motivate these problems in greater detail and describe how they are handled in PCC$^\theta$ at a high level. The introduction concludes with an outline of this document and its contributions.

1.1 Dependent Types and General Recursion

Dependently typed languages have usually held general recursion at arm's length. One reason is that unrestricted recursion makes it simple to write an infinite loop at any type. If every type is inhabited, the language is inconsistent when considered as a logic via the Curry-Howard isomorphism. Consistency is important—it ensures that the properties of programs we verify with dependent types really hold.

But programmers do not always stop to prove that their recursive functions terminate. Some intentionally write functions that loop or fail on certain inputs. A few programs, like web servers and REPLs, are meant to run forever. A language to support reasoning about practical programs must naturally handle these situations while retaining the property that its proofs are really proofs.

Worse, in languages like Coq and Agda [59, 49], termination arguments cloud the definitions of our programs. When reasoning about programs in these languages, we usually prefer to give our programs somewhat simple types and to verify properties about them after they have been defined. But this is not possible for termination, which must be verified as part of the function itself. As a result, we sometimes must rewrite functions in ways that are complex, inefficient and difficult to reason about in order to satisfy the termination checker. We would prefer a language that allows an *incremental* or *extrinsic* approach to termination, where we may reason about the termination behavior of functions after they have been defined (if ever).

To handle potentially non-terminating programs without sacrificing consistency, PCC$^\theta$ is broken into two fragments: a *programmatic* fragment with support for non-termination and a *logical* fragment that can reason about programs but is itself restricted for consistency. An important design goal for PCC$^\theta$ is *freedom of speech*—although the logical fragment is consistent, logical terms must be allowed to refer to non-logical terms, since we want to write proofs about programs. Informally, proofs may *talk about* programs, but not run them.

As a simple example, consider the following natural number division function written in a Haskell-like syntax:

```
prog div : Nat → Nat → Nat
div n m = if n < m then 0 else 1 + (div (n - m) m)
```

This function correctly computes the integer division of `n` by `m` unless `m` is `0`, in which case it loops forever. It is labeled "**`prog`**" to indicate that it must be defined in the programmatic fragment described above. Disappointingly, `div` cannot be written directly in popular dependently typed languages like Coq or Agda because it is not total.

There are many sensible properties a programmer might wish to verify about `div`. For example, `div 6 3` evaluates to `2` and `div n m ≤ n` when `m` is not zero. Even though `div` itself is defined in the programmatic fragment, we wish to state and prove these properties in the consistent logical fragment. For example:

```
  log div63 : div 6 3 = 2
  div63 = refl
```

Above, the program (aka proof) `div63` is tagged with "`log`" to indicate that it should be typechecked in the logical fragment. The proof itself is just reflexivity, based on the operational behavior of `div`.

To encourage incremental verification, PCC$^\theta$ also allows programs that are not known to be terminating to produce proofs. For example, programmers implementing a complicated decision procedure might begin by writing in the programmatic fragment and come back to prove termination at a later time. To support passing the proofs produced by such a procedure to the logical fragment, the language includes an *internalized logicality judgement*—programs may assert that other programs typecheck in a certain fragment. We use the new type form $A@\theta$, where θ is L or P for the logical or programmatic fragments, to claim that a term has type A in a particular fragment. For example, a SAT solver that is not known to be terminating might be given the following type:

```
  prog solver : (f : Formula) → Maybe ((Satisfiable f) @ L)
```

Here, `solver` takes in some representation of a formula and optionally produces a proof that it is satisfiable. Since `solver` is written in the programmatic fragment, it may not terminate. The `@L` in its type indicates that if it *does* return a value, that value typechecks in the logical fragment.

Calls to `solver` may not be evaluated directly in the logical fragment, since they may not terminate. However, if they do terminate, the logical fragment may analyze the resulting values. For example, it may pattern match on the result, as in the following fragment:

```
prog isSat : Maybe ((Satisfiable f) @ L)
isSat = solver f

log prf : ... f ...
prf = case isSat of
        Just y  → ... -- use proof y of satisfiability
        Nothing → ... -- use knowledge that solver returned Nothing
```

If the call to `solver` terminates, the logical fragment may use it to prove some property of the formula `f`.

1.2 Dependent Types and Equality

1.2.1 Traditional Intensional Equality

In traditional dependently typed languages like Coq and Agda, equality is defined as a datatype in the language.

```
data Eq {A : Type} (a : A) : A → Type where
  refl : Eq a a
```

The use of curly braces in the type of `Eq` indicates an implicit argument that the typechecker will infer for us based on the context. Typechecking for these languages is permitted to insert reduction anywhere, so although the definition above seems to say only two identical terms are equal, it actually says two terms are equal if both can be reduced to the same term. For example, in Agda we can prove that `refl` has the type `1+1 = 2`, because Agda can see that `1+1` reduces to `2`. Note that we use `=` here as syntax sugar; this equality could be written as `Eq (1+1) 2` instead.

One problem with this notion of equality can be seen in its definition—it demands that the terms being compared have the same type. This restriction can quickly become a nuisance when working with dependent types. As an example, consider the following datatype of length-indexed vectors and a corresponding append operation:

```
data Vec (A : Type) : Nat → Type where
  Nil  : Vec A 0
  Cons : {n : Nat} → A → Vec A n → Vec A (1+n)

app : {A : Type} → {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
app Nil         ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

The `app` function is, of course, associative. We might hope to prove this in the form of a program of type:

```
app_assoc : {A : Type} → {n m k : Nat}
          → (v1 : Vec A n) → (v2 : Vec A m) → (v3 : Vec A k)
          → (app v1 (app v2 v3)) = (app (app v1 v2) v3)
```

However, this type itself does not pass the typechecker in Coq or Agda. The problem is that the two vectors we wish to compare do not have the same type:

```
app v1 (app v2 v3)  :  Vec A (n + (m + k))
app (app v1 v2) v3  :  Vec A ((n + m) + k)
```

Since neither of `(n + (m + k))` and `((n + m) + k)` reduces to the other, the use of equality above is ill-typed. Thus, we cannot directly state the theorem that `app` is associative.

The solution is to explicitly prove that addition is also associative and to use this fact in the statement of the theorem above. We can prove the following theorem:

```
plus_assoc : (n m k : Nat) → (n + m) + k = n + (m + k)
```

However, a second inconvenience appears when we attempt to use this proof. The type systems of Coq and Agda have no way to automatically use proved equalities, so we must explicitly eliminate them. For example, the equality datatype above usually comes with an elimination principle like this one:

```
conv : {A : Type} → (f : A → Type) → {a b : A}
     → a = b → f a → f b
```

In some languages, `conv` is derived automatically as the elimination principle for the `Eq` type, while in other languages it is implemented by pattern matching. In either case, restating the `app_assoc` theorem in a way that passes the typechecker will require us to make explicit use of `conv`:

```
app_assoc : (A : Type) → (n m k : Nat)
          → (v1 : Vec A n) → (v2 : Vec A m) → (vs : Vec A k)
          →    (app v1 (app v2 v3))
              = conv (Vec A) (plus_assoc n m k)
                     (app (app v1 v2) v3)
```

This theorem no longer states quite what we wanted—we had hoped for an equality between `(app v1 (app v2 v3))` and `(app (v1 v2) v3)` and instead we are required to state an equality between the former and `conv` applied to the latter. This theorem does have the meaning that we want, because `conv` is operationally the identity function, but that meaning has been obscured by the unfortunate definition of equality. Worse, this theorem is inconvenient to use in other contexts because it mentions a *particular* proof that addition is associative, and the details of that other proof are now bound up in the type of our proof about vectors.

1.2.2 The PCC$^\theta$ Approach to Equality

A major design goal for PCC$^\theta$ is to prevent type information from getting in the way of equality reasoning. To that end, PCC$^\theta$ features a built-in equality type which is heterogeneous. That is, the terms related by equality do not need to have the same type. Thus, the property that vector append is associative may be stated directly, without a use of conversion.

Additionally, eliminations of equality proofs are unmarked in the syntax of PCC$^\theta$ terms. Thus, the proof of `app_assoc` is surprisingly simple:

```
log app_assoc : {A : Type} → {n m k : Nat}
              → (v1 : Vec A n) → (v2 : Vec A m) → (vs : Vec A k)
              → (app v1 (app v2 v3)) = (app (app v1 v2) v3)
app_assoc nil     v2 v3 = refl
app_assoc (x::xs) v2 v3 = refl
```

Intuitively, to typecheck this proof, we must do some equational reasoning about the definition of `app` and make a recursive call, `app_assoc xs v2 v3`, for an induction hypothesis. However, in PCC$^\theta$, uses of these derived equalities are not marked by an explicit call to a "`conv`" function in the syntax.

Of course, typechecking is undecidable for a language with these features. PCC$^\theta$ is not intended to be suitable for programming in or typechecking directly. In practice, programmers will need to add annotations to their programs to help the type-checker find the relevant equalities and places to use conversion in the above program. A complete surface language for PCC$^\theta$ is beyond the scope of this thesis. However, Vilhelm Sjöberg has designed such a language, Zombie [51], as part of his thesis. These two theses can be thought of as companion works, one examining the metatheory of PCC$^\theta$, and the other developing a surface language and novel type-inference algorithm that makes programming in PCC$^\theta$ practical. Intuitively, Zombie programs are elaborated into core PCC$^\theta$ derivations before typechecking.

Crucially, Zombie retains the pleasant equality we have described. To accomplish this, Zombie terms are *erased* into core terms before checking equalities or evaluation. This erasure operation removes any type annotations and uses of conversion. So, while writing `app_assoc` in Zombie requires some additional work (for example, an explicit appeal to the induction hypothesis), `app_assoc` is still given the type shown above and its implementation *erases* into this program.

1.3 Contributions and Outline

The remainder of this thesis is broken into chapters which each make several contributions:

- PCC$^\theta$ is essentially a predicative variant of the Calculus of Constructions [21] extended with the novel features outlined in this introduction. To demonstrate how these features solve the problems described above, it is useful to consider a somewhat larger language including convenient features like user-defined datatypes. Chapter 2 introduces such a language, Theta, and exhibits several example programs which involve non-termination and equality reasoning.

- The metatheory of PCC$^\theta$ is interesting and challenging. Demonstrating the type safety and consistency of a language with PCC$^\theta$-like fragments requires the development of a new technique, "partially step-indexed logical relations". Chapters 3 and 4 comprise a tutorial, including a demonstration that traditional techniques fail for languages like PCC$^\theta$ and examples of the new technique in the smaller settings of a simply typed language and a language with dependent types and equality, but not polymorphism or type level computation. The proofs in these two chapters have been mechanized in Coq and are available as part of a digital appendix to this thesis [13].

- Chapter 5 considers PCC$^\theta$ itself. This adds polymorphism and type level computation to the languages from Chapters 3 and 4 to achieve a "full spectrum" dependently typed core language. We examine its metatheory in great detail, extending the techniques of the previous chapters. This chapter proves type safety and the normalization and consistency of PCC$^\theta$'s logical fragment.

- Several of the features present in Theta but absent in PCC$^\theta$ have proved metatheoretically challenging. Chapter 6 demonstrates this by exhibiting a series of flawed language designs and failed proof techniques. In particular, PCC$^\theta$-like equality introduces subtle problems for traditional techniques in modeling dependently typed languages. While an exploration of these challenges is interesting in its own right, we hope that it is also useful as a roadmap for future work.

- Chapter 7 describes related work. Since this thesis tackles two problems, the related work falls into two broad categories—we compare PCC$^\theta$ both with other approaches to integrating non-termination with dependent types, and to other attempts to enhance dependently typed langauges with a more extensional equality. Finally, Chapter 8 concludes.

Chapter 2

The THETA Language

But they came, and when they finally made it here,
It was least that we could do to make our welcome clear.
Come on in. We haven't slept for weeks.
Drink some of this. This'll put color in your cheeks.

Color in Your Cheeks
The Mountain Goats

In this chapter we will formally specify a new dependently typed core language, THETA, and show several examples of programs written in it. This language makes concrete the solutions to the problems of general recursion and equality reasoning that were described in the introduction. In particular, THETA comprises a *logical fragment* that resembles Coq and Agda and a *programmatic fragment* that includes non-termination. Additionally, it includes a built-in equality type that ignores type information and whose eliminations are unmarked in expressions.

THETA is an extension of PCC$^\theta$ with features like user-defined datatypes and an infinite hierarchy of universes. These features are typically convenient in dependently typed languages (and thus useful in our examples), but they are too syntactically burdensome or metatheoretically challenging to include in the language our formal proofs consider. PCC$^\theta$ itself is examined in Chapter 5, and the differences between PCC$^\theta$ and THETA are the subject of Chapter 6.

Typechecking for THETA expressions is undecidable. One reason is that the equality relation itself is undecidable—since programs in THETA may not terminate, the traditional "normalize and compare" approach will not work here. Additionally, uses of equality are unmarked in the syntax of expressions. So, even if equality were decidable, the typechecker would need to "guess" where to insert uses of it. In this sense, THETA's notion of equality is related to extensional type theories, like Nuprl [18]. This connection is considered in greater detail in Chapter 7.

As described in Chapter 1, THETA programs are written in the surface language ZOMBIE, elaborated into THETA derivations for typechecking, and then erased into

Expressions

$$
\begin{array}{lll}
a,\ b,\ A,\ B & ::= & \mathsf{Type}_\ell \mid (x : A) \to\ B \mid a = b \mid D\,\overline{A_i}^{\,i} \mid a < b \mid A@\theta \\
& & \mid x \mid \lambda x.\, b \mid \mathsf{rec}\, f\ x.b \mid \mathsf{ind}\, f\ x.b \mid b\ a \mid \mathsf{refl} \mid d\,\overline{a_i}^{\,i} \mid \mathsf{abort} \mid \mathsf{contra} \\
& & \mid \mathsf{ord} \mid \mathsf{case}\ a\ \mathsf{of}\ \{\ \overline{d_i\,\overline{x}_i \Rightarrow a_i}^{\,i}\ \}
\end{array}
$$

Consistency Classifiers

$$
\theta \quad ::= \quad \mathsf{L} \mid \mathsf{P}
$$

Contexts

$$
\Gamma \quad ::= \quad . \mid \Gamma, x :^{\theta} A \mid \Gamma, \mathsf{data}\ D\,\Delta : A\ \mathsf{where}\ \{\ \overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i}\ \} \mid \Gamma, \mathsf{data}\ D\,\Delta : A
$$

Telescopes

$$
\Delta \quad ::= \quad \cdot \mid (x : A)\Delta
$$

Values

$$
\begin{array}{lll}
v & ::= & \mathsf{Type}_\ell \mid (x : A) \to\ B \mid a = b \mid D\,\overline{A_i}^{\,i} \mid a < b \mid A@\theta \\
& & \mid x \mid \lambda x.\, a \mid \mathsf{rec}\, f\ x.a \mid \mathsf{ind}\, f\ x.a \mid \mathsf{refl} \mid d\,\overline{v_i} \mid
\end{array}
$$

Figure 2.1: THETA: Syntax

THETA expressions for evaluation. While describing the syntax and static semantics for THETA, we will point out several places where allowing undecidable typechecking makes it possible to drop typing information from terms and thus provide a more convenient notion of equality.

2.1 Syntax and Operational Semantics

The abstract syntax of THETA is given in Figure 2.1. For uniformity, terms and types are collapsed into one syntactic category, as in the presentation of the lambda cube [6] and many other dependently typed languages [59, 49, 42]. As a matter of discipline, we will use the word "expression" to refer to any element of this grammar, reserving "term" for expressions at the term level and "type" for expressions at the type level. When possible, we will use lowercase metavariables for terms and uppercase metavariables for types.

The first line of the grammar exhibits the language's types. Functions are classified by dependent pi types $(x : A) \to\ B$. We will sometimes write $A \to B$ as syntax sugar for $(x : A) \to\ B$ when x does not occur free in B. THETA includes a built-in equality type $a = b$ and user-defined datatypes $D\,\overline{A_i}^{\,i}$. The type form $a < b$ is an ordering used for terminating recursion and the new type $A@\theta$ internalizes the consistency classifier portion of the typing judgement. Types are classified by the kind Type_ℓ where $\ell \in \mathbb{N}$,

Evaluation Contexts

$$\mathcal{E} \quad ::= \quad [\cdot] \mid \mathcal{E}\; b \mid v\; \mathcal{E} \mid d\; \overline{v_i}^{\,i}\; \mathcal{E}\; \overline{a_j}^{\,j} \mid \mathsf{case}\, \mathcal{E}\; \mathsf{of}\; \{\; \overline{d_i\, \Delta_i \Rightarrow a_i}^{\,i}\; \}$$

$$\boxed{a \rightsquigarrow b}$$

$$\frac{}{(\lambda x.\, b)\; v \rightsquigarrow [v/x]b}\;\textsc{SLam} \qquad \frac{}{(\mathsf{rec}\, f\; x.b)\; v \rightsquigarrow [v/x][\mathsf{rec}\, f\; x.b/f]b}\;\textsc{SRec}$$

$$\frac{}{(\mathsf{ind}\, f\; x.b)\; v \rightsquigarrow [v/x][\lambda x.\, \lambda z.\, (\mathsf{ind}\; f\; x.b)\; x/f]b}\;\textsc{SInd}$$

$$\frac{}{\mathsf{case}\; d_k\; \overline{v_j}^{\,j}\; \mathsf{of}\; \{\; \overline{d_i\, \Delta_i \Rightarrow a_i}^{\,i}\; \} \rightsquigarrow [\;\overline{v_j}^{\,j}\;/\Delta_k]a_k}\;\textsc{SCase}$$

$$\frac{}{\mathcal{E}[\mathsf{abort}] \rightsquigarrow \mathsf{abort}}\;\textsc{SAbort} \qquad \frac{a \rightsquigarrow b}{\mathcal{E}[a] \rightsquigarrow \mathcal{E}[b]}\;\textsc{SCtx}$$

$$\boxed{a \rightsquigarrow^* b}$$

$$\frac{}{a \rightsquigarrow^* a}\;\textsc{MSRefl} \qquad \frac{\begin{array}{c} a \rightsquigarrow b \\ b \rightsquigarrow^* b' \end{array}}{a \rightsquigarrow^* b'}\;\textsc{MSStep}$$

Figure 2.2: Theta: Operational Semantics

representing an infinite hierarchy of universes.

In addition to lambdas, the language includes two types of recursive functions: $\mathsf{rec}\, f\; x.b$ is used for unrestricted general recursion, while $\mathsf{ind}\, f\; x.b$ is used for terminating recursion. Theta also includes abort, which is used for general failure (similar to `error` in Haskell), contra, which will be used to eliminate contradictory equalities, and ord, which constructs proofs of $a < b$ to be used with terminating recursion, as we will see below.

The language uses a *consistency classifier* θ to distinguish the two fragments—L for logical and P for programmatic. Contexts Γ record the logicality of each variable. Contexts also include the declarations of user-defined datatypes. The second datatype form, $\mathsf{data}\, D\, \Delta : A$, specifies abstract datatypes without a definition, and is used only internally when checking datatype constructors for well-formedness.

These datatype declarations mention *telescopes* Δ. A telescope is a list of typed variables, $(x_1 : A_1) \ldots (x_j : A_j)$. We will sometimes abuse telescope notation by using Δ as a list of variables, as in the constructor application $d\, \Delta$ or the multi-substitution $[\,\overline{a_i}^{\,i \in 1..j}\,/\Delta]b$. We will also write $(\Gamma, \Delta\, \theta)$ to indicate the context Γ extended with the variable bindings in Δ where each has been tagged with the consistency classifier θ.

Figure 2.2 provides the operational semantics of Theta. Call-by-value reduction $a \rightsquigarrow b$ is specified as beta rules for the language's elimination forms closed over evaluation contexts $\mathcal{E}$. The reduction rules are standard except for beta reduction of

$$\boxed{a \Rrightarrow b}$$

$$\frac{}{a \Rrightarrow a}\ \textsc{PRefl} \qquad \frac{v \Rrightarrow v' \quad b \Rrightarrow b'}{(\lambda x.\, b)\ v \Rrightarrow [v'/x]b'}\ \textsc{PLam}$$

$$\frac{b \Rrightarrow b' \quad v \Rrightarrow v'}{(\mathsf{rec}\ f\ x.b)\ v \Rrightarrow [v'/x][\mathsf{rec}\ f\ x.b'/f]b'}\ \textsc{PRec}$$

$$\frac{b \Rrightarrow b' \quad v \Rrightarrow v'}{(\mathsf{ind}\ f\ x.b)\ v \Rrightarrow [v'/x][\lambda x.\, \lambda z.\, (\mathsf{ind}\ f\ x.b')\ x/f]b'}\ \textsc{PInd}$$

$$\frac{\overline{v_j \Rrightarrow v'_j}^{\,j} \quad a_k \Rrightarrow a'_k}{\mathsf{case}\ d_k\ \overline{v_j}^{\,j}\ \mathsf{of}\ \{\ \overline{d_i\,\Delta_i \Rightarrow a_i}^{\,i}\ \} \Rrightarrow [\ \overline{v'_j}^{\,j}\ /\Delta_k]a'_k}\ \textsc{PCase} \qquad \frac{}{\mathcal{E}[\mathsf{abort}] \Rrightarrow \mathsf{abort}}\ \textsc{PAbort}$$

$$\frac{\overline{a_j \Rrightarrow a'_j}^{\,j}}{d\ \overline{a_j}^{\,j} \Rrightarrow d\ \overline{a'_j}^{\,j}}\ \textsc{PTrmCon1} \qquad \frac{\overline{a_j \Rrightarrow a'_j}^{\,j}}{D\ \overline{a_j}^{\,j} \Rrightarrow D\ \overline{a'_j}^{\,j}}\ \textsc{PTypCon1}$$

$$\frac{b \Rrightarrow b'}{\lambda x.\, b \Rrightarrow \lambda x.\, b'}\ \textsc{PLam1} \qquad \frac{b \Rrightarrow b'}{\mathsf{rec}\ f\ x.b \Rrightarrow \mathsf{rec}\ f\ x.b'}\ \textsc{PRec1}$$

$$\frac{b \Rrightarrow b'}{\mathsf{ind}\ f\ x.b \Rrightarrow \mathsf{ind}\ f\ x.b'}\ \textsc{PInd1} \qquad \frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{b\ a \Rrightarrow b'\ a'}\ \textsc{PApp1}$$

$$\frac{A \Rrightarrow A'}{A@\theta \Rrightarrow A'@\theta}\ \textsc{PAt1} \qquad \frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{(x : A) \to B \Rrightarrow (x : A') \to B'}\ \textsc{PArr1}$$

$$\frac{a \Rrightarrow a' \quad \overline{b_i \Rrightarrow b'_i}^{\,i}}{\mathsf{case}\ a\ \mathsf{of}\ \{\ \overline{d_i\,\overline{x}_i \Rightarrow b_i}^{\,i}\ \} \Rrightarrow \mathsf{case}\ a'\ \mathsf{of}\ \{\ \overline{d_i\,\overline{x}_i \Rightarrow b'_i}^{\,i}\ \}}\ \textsc{PCase1} \qquad \frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{a = b \Rrightarrow a' = b'}\ \textsc{PEq1}$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{a < b \Rrightarrow a' < b'}\ \text{PLT1}$$

$$\boxed{a \Rrightarrow^{*} b}$$

$$\frac{}{a \Rrightarrow^{*} a}\ \textsc{MPRefl} \qquad \frac{\begin{array}{c} a \Rrightarrow b \\ b \Rrightarrow^{*} b' \end{array}}{a \Rrightarrow^{*} b'}\ \textsc{MPStep}$$

Figure 2.3: THETA: Parallel Reduction

terminating recursion, which inserts a surprising eta-expansion. This will be explained in Section 2.2 when we describe the corresponding typing rule.

We also require a notion of parallel reduction for THETA. This appears in Figure 2.3. It permits reductions under binders and will be used when checking equalities between terms. Call-by-value reduction is a subrelation of parallel reduction.

2.2 Typing

In this section we describe THETA's typing relation. The typing judgement is indexed by a consistency classifier θ to indicate in which fragment the term is being checked:

$$\Gamma \vdash^{\theta} a : A$$

The use of unsafe features like general recursion and abort is allowed only when θ is P. Most other typing rules for terms are generic in θ, and the fragments can explicitly interact in a few ways, as we will see in Section 2.2.2. For ease of explanation, we have divided the typing judgement into groups of related rules below.

2.2.1 Variables, Universes and Functions

Figure 2.4 presents the typing rules related to variables, universes, and functions. Except for the consistency classifiers and the presence of general recursion, these rules resemble those of other dependently typed core languages, like the Calculus of Constructions [21]. For example, in rule TVAR we see that variables are tagged with a consistency classifier in the context. This indicates whether the value assigned to this variable is guaranteed to typecheck in the logical fragment. This rule includes the premise $\vdash \Gamma$ which checks contexts for well-formedness. It is defined in Section 2.2.4.

Rules TTYPE and TCUMUL handle the cumulative hierarchy of universes. Intuitively, the type system is divided into an infinite number of levels. Terms are at the base level, and they are classified by types. Types themselves are classified by the kind Type_0, which has kind Type_1 and so on. Rule TCUMUL makes this hierarchy "cumulative", in that any expression which checks at level ℓ will also check at all higher levels.

Rule TARR checks arrow types. All expressions which can classify other expressions have the type Type_ℓ for some level ℓ. In this case ℓ is the maximum of the levels of the arrow's domain and range. This enforces *predicative* polymorphism—a type never quantifies over itself. To see this, observe that polymorphism occurs when the domain of an arrow type is Type_ℓ (i.e., when a type quantifies over other types). But since we have already seen that Type_ℓ has the type $\mathsf{Type}_{\ell+1}$, such an arrow type would live at level $\ell + 1$ or higher and thus not be in its own domain. The additional premise $\mathsf{Mob}\,(A)$ demands that the domain of a function type is always "mobile" and will be explained in Section 2.2.2.

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{l}\vdash \Gamma \\ x :^{\theta} A \in \Gamma\end{array}}{\Gamma \vdash^{\theta} x : A}\ \textsc{TVar} \qquad \frac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Type}_{\ell} : \mathsf{Type}_{\ell+1}}\ \textsc{TType} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} b : \mathsf{Type}_{\ell} \\ \ell < \ell'\end{array}}{\Gamma \vdash^{\theta} b : \mathsf{Type}_{\ell'}}\ \textsc{TCumul}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_{\ell_1} \quad \mathsf{Mob}\,(A) \\ \Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} B : \mathsf{Type}_{\ell_2}\end{array}}{\Gamma \vdash^{\mathsf{L}} (x : A) \to\ B : \mathsf{Type}_{(\max(\ell_1,\ell_2))}}\ \textsc{TArr} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} b : (x : A) \to\ B \\ \Gamma \vdash^{\theta} a : A \\ \Gamma \vdash^{\mathsf{L}} [a/x]B : \mathsf{Type}_{\ell}\end{array}}{\Gamma \vdash^{\theta} b\ a : [a/x]B}\ \textsc{TApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} (x : A) \to\ B : \mathsf{Type}_{\ell} \\ \Gamma, x :^{\theta} A \vdash^{\theta} b : B\end{array}}{\Gamma \vdash^{\theta} \lambda x.\, b : (x : A) \to\ B}\ \textsc{TLam} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} (y : A) \to\ B : \mathsf{Type}_{\ell} \\ \Gamma, y :^{\mathsf{P}} B, f :^{\mathsf{P}} (y : A) \to\ B \vdash^{\mathsf{P}} b : B\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\ y.b : (y : A) \to\ B}\ \textsc{TRec}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} (y : A) \to\ B : \mathsf{Type}_{\ell} \\ \Gamma, y :^{\mathsf{L}} A, f :^{\mathsf{L}} (x : A) \to\ (z : x < y) \to\ [x/y]B \vdash^{\mathsf{L}} b : B\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ind}\, f\ y.b : (y : A) \to\ B}\ \textsc{TInd}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \quad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a < b : \mathsf{Type}_0}\ \textsc{TSmaller} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a : b = d_1\ \overline{b_i}^{\,i \in 1\ldots j}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ord} : b_i < b}\ \textsc{TOrd}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : b_1 < b_2 \\ \Gamma \vdash^{\mathsf{L}} a' : b_2 < b_3\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ord} : b_1 < b_3}\ \textsc{TOrdTrans} \qquad \frac{\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_{\ell}}{\Gamma \vdash^{\mathsf{P}} \mathsf{abort} : A}\ \textsc{TAbort}$$

Figure 2.4: THETA Typing: Variables, Universes and Functions

Note that for an arrow type to typecheck, its components must check in fragment L. In general, we will maintain the invariant that any expression used to classify other expressions must check in the logical fragment. This prevents non-termination at the type level. The reasons for this restriction are described in Section 6.1.

The rule for function application, TApp, differs from the usual application rule in pure dependently-typed languages in the additional premise $\Gamma \vdash^{\mathsf{L}} [a/x]B : \mathsf{Type}_{\ell}$. This checks that the result type is well-formed. Some rules of the language are sensitive to whether expressions are values (such as the Box rules in Section 2.2.2 and β-reduction which occurs when checking equality). Because values include variables, substituting an expression a for the variable x could violate a value restriction that is allowing B to typecheck, necessitating this extra premise.

Any dependently typed language that combines pure and effectful code will likely have to restrict the application rule in some way. Previous work [39, 34, 57] is more restrictive in typing applications. These systems use two rules: one which permits only *value dependency* and requires the argument to be a value, and one which allows a non-dependent function to be applied to an arbitrary argument.

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash f\ v : [v/x]B} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

Since substituting a value can never violate a value restriction in B, our application rule subsumes the value-dependent version. Likewise, in the case of no dependency, the extra premise can never fail because the substitution has no effect on B. As we will see in the examples below, being able to call dependent functions with non-value arguments is useful in many situations. For example, in the introduction we used the example of proving that the append function for vectors is associative. There, append must be applied to a vector whose length has the form `(n + m)`, a non-value.[1]

There are three ways to define functions in THETA. Rule TLAM allows non-recursive functions in either fragment, whereas rule TREC allows general recursive rec-expressions and can only be used in the programmatic fragment. Note that the consistency classifier does not simply describe whether an expression terminates—rec $f\ x.b$ is a normal form, but rule TREC it is still restricted to fragment P. Intuitively, this restriction exists because functions defined in this way cannot be thought of as constructive proofs.

Additionally, terminating recursion is provided in the logical fragment by rule TIND. When typechecking the body of a terminating recursive function (ind $f\ x.b$), the recursive call f takes an extra argument proving that it is being applied to a "smaller" value than the initial argument x. This ensures termination. When beta-reducing such an expression, this argument is ignored by wrapping the function in an extra lambda (rule SIND from Figure 2.2).

The definition of "smaller" comprises rules TSMALLER, TORD and TORDTRANS. The first of these rules says that $a < b$ is a type for any well-typed a and b. As we will see in Section 2.2.2, expressions that check in L also check in P, so the use of the programmatic fragment in the premises of this rule is not a restriction. Rule TORD says that an expression b_i is smaller than b if there is a proof that b is a term constructor applied to b_i (using the built-in equality type that will be specified in Section 2.2.3). Thus, the use of $<$ in the TIND rule captures structural recursion—we are allowed to make recursive calls when the argument is a piece of the original input obtained by pattern matching. Rule TORDTRANS makes the $<$ relation transitive, which allows deeper patterns.

[1]Since we made the length arguments to `app` implicit, `(n + m)` does not appear in the statement of the associativity result. But implicit arguments are a surface language feature—this example would be elaborated into a core term where the length argument was explicit. Since `(n + m)` is a non-value and the length does appear in the result type of `app`, the restricted rules shown here would not work.

2.2.2 The Fragments

$$\boxed{\mathsf{Mob}\,(A)}$$

$$\frac{}{\mathsf{Mob}\,(a = b)}\ \textsc{MobEq} \qquad \frac{}{\mathsf{Mob}\,(a < b)}\ \textsc{MobSmaller} \qquad \frac{}{\mathsf{Mob}\,(B@\theta)}\ \textsc{MobAt}$$

$$\frac{}{\mathsf{Mob}\,(D\,\overline{A_i}^{\,i})}\ \textsc{MobData} \qquad \frac{}{\mathsf{Mob}\,(\mathsf{Type}_\ell)}\ \textsc{MobType}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\ \textsc{TSub} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{P}} v : A\\ \mathsf{Mob}\,(A)\end{array}}{\Gamma \vdash^{\mathsf{L}} v : A}\ \textsc{TMVal}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_\ell}{\Gamma \vdash^{\mathsf{L}} A@\theta : \mathsf{Type}_\ell}\ \textsc{TAt} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{L}} a : A@\theta'}\ \textsc{TBoxLL}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\ \textsc{TBoxLV} \qquad \frac{\Gamma \vdash^{\theta} a : A}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}\ \textsc{TBoxP} \qquad \frac{\Gamma \vdash^{\theta} v : B@\theta'}{\Gamma \vdash^{\theta'} v : B}\ \textsc{TUnboxVal}$$

Figure 2.5: THETA Typing: The Fragments

Figure 2.5 shows typing rules allowing implicit and explicit interactions between the fragments. One such interaction is that every logical expression can be safely used programmatically. We reflect this fact into the type system via the "subsumption" rule TSUB, which says that if an expression a typechecks logically, then it also typechecks programmatically. For example, a logical term can always be supplied to a function expecting a programmatic argument, and a function defined in the logical fragment can be freely used in the programmatic fragment. This rule is useful to avoid code duplication.

Subsumption also eliminates duplication in the design of the language. For example, we need only one type $a = b$ to describe equalities between logical expressions or programmatic expressions. In fact, we can also equate expressions from different fragments.

Internalized Consistency Classification

To provide a general mechanism for logical expressions to appear in programs and programmatic values to appear in proofs, we introduce a type that internalizes the typing judgment, written $A@\theta$. Nonterminating programs can take logical proofs as preconditions (with functions of type $(x : A@\mathsf{L}) \to B$), return them as postconditions (with functions of type $(x : A) \to (B@\mathsf{L})$), and store them in data structures (for example, with lists of type $\mathsf{List}\,(A@\mathsf{L})$). At the same time, logical lemmas can use @ to manipulate values from the programmatic fragment.

The rules for the $A@\theta$ type appear in Figure 2.5. Intuitively, the judgment $\Gamma \vdash^{\theta_1} a : A@\theta_2$ holds if the fragment θ_1 may safely observe that $\Gamma \vdash^{\theta_2} a : A$. This intuition is captured by the three introduction rules. The programmatic fragment can internalize any typing judgement (TBoxP), but in the logical fragment (TBoxLL and TBoxLV) we sometimes need a value restriction to ensure termination. Therefore, rule TBoxLV only applies when the subject of the typing rule is a value. The rule TBoxL can introduce $A@\theta$ for any θ since logical terms are also programmatic. Both introduction and elimination of @ is unmarked in the syntax, so the reduction behavior of an expression is unaffected by whether the type system deems it to be provably terminating or not.

As an example, a recursive function f can require an argument to be a proof by marking it $@\mathsf{L}$, e.g., $A@\mathsf{L} \to B$, forcing that argument to be checked in fragment L:

$$\frac{\Gamma \vdash^{\mathsf{P}} f : A@\mathsf{L} \to B \qquad \dfrac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A@\mathsf{L}}\,\textsc{TBoxP}}{\Gamma \vdash^{\mathsf{P}} f\ a : B}\,\textsc{TApp}$$

Similarly, a logical lemma g can be applied to a programmatic value by marking it $@\mathsf{P}$:

$$\frac{\Gamma \vdash^{\mathsf{L}} g : A@\mathsf{P} \to B \qquad \dfrac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\,\textsc{TBoxLV}}{\Gamma \vdash^{\mathsf{L}} g\ v : B}\,\textsc{TApp}$$

Of course, g can only be defined in the logical fragment if it is careful to not use its argument in unsafe ways. For example, using TConv we can prove a lemma of type

```
(n: Nat) → (f: (Nat → Nat) @ P) → (f (plus n 0) = f n)
```

because reasoning about `f` does not require calling `f` at runtime.

There is no way to apply a logical lemma to a programmatic *non*-value expression. If an expression `a` may diverge, then so may `f a`, so we must not assign it a type in the logical fragment.[2] However, we can work around this restriction either by first evaluating `a` to a value in the programmatic fragment or by thunking.

The @-types are eliminated by the rule TUnboxVal. To preserve termination, the rule is restricted to apply only to values. We believe it is possible to extend the system with three "unbox" rules, the symmetric twins of our three "box" rules, but since TUnboxVal has been sufficient for our examples, we have not yet explored this possibility in detail.

Recall, from the introduction, the function `solver`:

```
prog solver : (f:Formula) → Maybe ((Satisfiable f) @ L)
```

[2]This is one drawback of working in a strict rather than a lazy language. If we know that `f` is nonstrict, then this application is indeed safe.

In the introduction, we asserted that the following code typechecks.

```
prog isSat : Maybe ((Satisfiable f) @ L)
isSat = solver f

log prf : ... f ...
prf = case isSat of
        Just y  → ... -- use proof y of satisfiability
        Nothing → ... -- use knowledge that solver returned Nothing
```

In this example, the logical program `prf` cannot directly treat `solver f` as a proof because it may diverge. However, once it has been evaluated to a value, it can safely be used by the logical fragment. Defining the intermediate variable `isSat` forces evaluation of the expression `solver f`, introducing a new programmatic variable of type `Maybe ((Satisfiable f) @ L)` into the context. Because variables are values, any logical context can freely use the variable through TUnboxVal, even though it was computed by the programmatic language.

Mobile Types

The consistency classifier tracks which *expressions* are known to come from a consistent language. For some types of *values*, however, the rules described so far can be unnecessarily conservative. For example, while a programmatic expression of type Nat may diverge, a programmatic value of that type is just a number, so we can treat it as if it were logical. On the other hand, we cannot treat a programmatic function value as logical, since it might cause non-termination when applied.

The rule TMVal (Figure 2.5) allows values to be moved from the programmatic to the logical fragment. It relies on an auxiliary judgment $\mathsf{Mob}\,(A)$. Intuitively, a type is mobile if the same set of values inhabit the type when $\theta = \mathsf{L}$ and when $\theta = \mathsf{P}$. In particular, these types do not include functions (though any type may be made mobile by tagging its fragment with @).

Concretely, the equality and less-than types are mobile as they are inhabited only by refl and ord, respectively. Any @-type is mobile, since it fixes a particular θ independent of the one on the typing judgment. Datatypes are also mobile—as we will see in Section 2.2.4, data contained in a datatype must be mobile, so the whole datatype is as well. Finally, Type_ℓ is mobile because all types check logically in Theta.

The arguments to functions must always have mobile types. This prevents problems when functions are moved between the fragments. For example, when a function is defined in the logical fragment, the body of the function assumes that its argument checks logically. If this function is then moved to the programmatic argument via subsumption and applied to a programmatic argument, the body of the function may no longer be sensible. To solve this problem, TArr requires the domains of function types to be mobile. One consequence of this restriction is that higher-order functions must use @-types to specify which fragment their arguments belong to. For example,

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \qquad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a = b : \mathsf{Type}_0}\ \textsc{TEq} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a_1 = a_2 : \mathsf{Type}_\ell \qquad a_1 \Rrightarrow^* b \quad a_2 \Rrightarrow^* b}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a_1 = a_2}\ \textsc{TRefl}$$

$$\frac{\Gamma \vdash^{\theta} a : [b_1/x]A \qquad \Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \qquad \Gamma \vdash^{\mathsf{L}} [b_2/x]A : \mathsf{Type}_\ell}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \textsc{TConv} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a : A_1 = A_2 \qquad \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_\ell \qquad \mathsf{hd}(A_1) = \mathit{hf}_1 \quad \mathsf{hd}(A_2) = \mathit{hf}_2 \quad \mathit{hf}_1 \neq \mathit{hf}_2}{\Gamma \vdash^{\mathsf{L}} \mathsf{contra} : A}\ \textsc{TContra}$$

Head Forms

$$\mathit{hf} \quad ::= \quad \mathsf{HType}\,\ell \mid \mathsf{HData}\,D \mid \mathsf{HAt}\,\theta \mid \mathsf{HArr} \mid \mathsf{HEq}$$

$$\boxed{\mathsf{hd}(A) = \mathit{hf}}$$

$$\frac{}{\mathsf{hd}(\mathsf{Type}_\ell) = \mathsf{HType}\,\ell}\ \textsc{HType} \qquad \frac{}{\mathsf{hd}(D\,\overline{A_i}^{\,i}) = \mathsf{HData}\,D}\ \textsc{HData}$$

$$\frac{}{\mathsf{hd}(A@\theta) = \mathsf{HAt}\,\theta}\ \textsc{HAt} \qquad \frac{}{\mathsf{hd}((x : A) \to B) = \mathsf{HArr}}\ \textsc{HArr} \qquad \frac{}{\mathsf{hd}(a = b) = \mathsf{HEq}}\ \textsc{HEq}$$

Figure 2.6: THETA Typing: Equality

the type `(Nat → Nat) → A` is not well-formed, so the programmer must choose either `((Nat → Nat) @ L) → A` or `((Nat → Nat) @ P) → A`.

The use of mobile rules allow programmers to write simpler types because mobile types never need to be tagged with logical classifiers. For example, without loss of generality we can give a function the type `(a = b) → B` instead of `((a = b) @ L) → B`, since, when needed, the body of the function can treat the argument as logical through TMVAL. Similarly, multiple @'s have no effect beyond the innermost @ in a type. Values of type `A @ P @ L @ P @ L` can be used as if they had type `A @ P`.

2.2.3 Equality

A major goal for THETA is the ability to write proofs about potentially non-terminating programs. For example, in the introduction we considered several properties of a non-total division function. Our rules for propositional equality (Figure 2.6) are designed to support such reasoning uniformly, based only on the run-time behavior of the expressions being equated, and independently of the fragment in which they are defined. Therefore, the rule TEQ shows that the type $a = b$ is well-formed and in the logical fragment even when a and b can be typechecked only programmatically.

This is freedom of speech: proofs can refer to nonterminating programs.

The term refl is the primitive proof of equality. Rule TREFL says that refl is a proof of $a = b$ just when a and b reduce to a common expression. The notion of reduction used in the rule is *parallel reduction*, written $a \Rrightarrow b$. This relation extends ordinary evaluation $a \rightsquigarrow b$ by allowing reduction under binders, e.g. $(\lambda x.1+1) \Rrightarrow (\lambda x.2)$ even though $(\lambda x.1+1)$ is already a value. Having this extra flexibility makes equality more expressive.

Equalities are used to modify the type assigned to an expression via the elimination rule TCONV. We demand that the equality proof used in conversion typechecks in the logical fragment for type safety. All types are inhabited in the programmatic fragment, so if we permitted the user to convert using a programmatic proof of, say, $\mathsf{Nat} = \mathsf{Nat} \to \mathsf{Nat}$, it would be easy to create a stuck term. Similar to TAPP, we need to check that b_2 does not violate any value restrictions, so the last premise checks the well-formedness of the type given to the converted term. Rule TCONV is quite general, and may be used to change some small part of A or the entire type by picking x for A.

Uses of TCONV are not marked in the term because they are not relevant at runtime. Again, types should describe terms without interfering with equality; we do not want terms with the same runtime behavior to be considered unequal due to uses of conversion. This treatment of equality is a variant of Sjöberg *et al.* [52]. However, that setting did not include a logical sublanguage; instead it enforced soundness by requiring the proof term used in conversion to be a value.

Finally, rule TCONTRA may be used to eliminate contradictory equalities. An equality is considered a contradiction if it equates two types with different *head forms*, as described in Figure 2.6. For example, in the presence of a logical proof that $\mathsf{Type}_0 = (\mathsf{Nat} \to \mathsf{Nat})$, the term contra may be given any type. The notion of head forms given here considers only types, but it is possible to derive within THETA a similar elimination principle for equalities between distinct data constructors, like $\mathsf{true} = \mathsf{false}$. The ZOMBIE implementation includes this facility.

2.2.4 Datatypes

The rules for datatypes and pattern matching appear in Figure 2.7. Datatype declarations have the form:

$$\mathsf{data}\ D\,\Delta : \mathsf{Type}_\ell\ \mathsf{where}\ \{\ \overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1..k}\ \}$$

Here, D is the type being introduced. The telescope Δ contains its parameters. Its term constructors are the d_i and their arguments are Δ_i.

Datatype declarations are checked by the context well-formedness judgement $\vdash \Gamma$ (Figure 2.7). Its rule CDATA handles datatypes. The first two premises use the auxiliary judgement $\Gamma \vdash \Delta : \mathsf{Type}_\ell$ to ensure that the types of the type constructor and term constructors are themselves well typed and reside at the appropriate universe

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{l}\mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell}\ \mathsf{where}\ \{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1...j}\,\} \in \Gamma \\ \Gamma \vdash^{\mathsf{L}} \overline{A_i}^{\,i} : \Delta \quad \vdash \Gamma\end{array}}{\Gamma \vdash^{\mathsf{L}} D\,\overline{A_i}^{\,i} : \mathsf{Type}_{\ell}}\ \textsc{TTCon} \qquad \frac{\begin{array}{l}\mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell} \in \Gamma \\ \Gamma \vdash^{\mathsf{L}} \overline{A_i}^{\,i} : \Delta \quad \vdash \Gamma\end{array}}{\Gamma \vdash^{\theta} D\,\overline{A_i}^{\,i} : \mathsf{Type}_{\ell}}\ \textsc{TATCon}$$

$$\frac{\begin{array}{l}\mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell}\ \mathsf{where}\ \{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1...j}\,\} \in \Gamma \\ \Gamma \vdash^{\mathsf{L}} \overline{A_i}^{\,i} : \Delta \quad \vdash \Gamma \\ \Gamma \vdash^{\theta} \overline{a_i}^{\,i} : [\overline{A_i}^{\,i}/\Delta]\Delta_k\end{array}}{\Gamma \vdash^{\theta} d_k\,\overline{a_i}^{\,i} : D\,\overline{A_i}^{\,i}}\ \textsc{TDCon}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : D\,\overline{a_i}^{\,i} \quad \Gamma \vdash^{\mathsf{L}} B : \mathsf{Type}_{\ell_2} \\ \mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell_1}\ \mathsf{where}\ \{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1...k}\,\} \in \Gamma \\ \forall i \in 1...k,\quad \Gamma, [\overline{a_i}^{\,i}/\Delta]\Delta_i\,\theta, y :^{\mathsf{L}} a = d_i\,\Delta_i \vdash^{\theta} b_i : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{case}\ a\ \mathsf{of}\ \{\,\overline{d_i\,\Delta_i \Rightarrow b_i}^{\,i\in 1...k}\,\} : B}\ \textsc{TCase}$$

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

$$\frac{}{\vdash \cdot}\ \textsc{CNil}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_{\ell} \\ x \notin \mathsf{dom}\,(\Gamma) \quad \vdash \Gamma\end{array}}{\vdash \Gamma, x :^{\theta} A}\ \textsc{CVar} \qquad \frac{\begin{array}{l}\Gamma \vdash \Delta : \mathsf{Type}_{\ell} \\ \forall i \in 1...k,\quad \Gamma, \mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell}, \Delta\,\mathsf{L} \vdash \Delta_i : \mathsf{Type}_{\ell} \\ \forall i \in 1...k,\quad D\text{'s occurances in } \Delta_i \text{ are strictly positive} \\ D \notin \mathsf{dom}\,(\Gamma) \quad \overline{d_i \notin \mathsf{dom}\,(\Gamma)}^{\,i\in 1...k} \quad \vdash \Gamma\end{array}}{\vdash \Gamma, \mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell}\ \mathsf{where}\ \{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1...k}\,\}}\ \textsc{CData}$$

$$\boxed{\Gamma \vdash \Delta : \mathsf{Type}_{\ell}}$$

$$\frac{}{\Gamma \vdash \cdot : \mathsf{Type}_{\ell}}\ \textsc{TeleWFNil} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_{\ell} \quad \mathsf{Mob}\,(A) \\ \Gamma, x :^{\mathsf{L}} A \vdash \Delta : \mathsf{Type}_{\ell}\end{array}}{\Gamma \vdash (x : A)\Delta : \mathsf{Type}_{\ell}}\ \textsc{TeleWFCons}$$

$$\boxed{\Gamma \vdash^{\theta} \overline{a_i}^{\,i} : \Delta}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash^{\theta} \cdot : \cdot}\ \textsc{TeleNil} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} a : A \\ \Gamma \vdash^{\theta} \overline{a_i}^{\,i} : [a/x]\Delta\end{array}}{\Gamma \vdash^{\theta} a\,\overline{a_i}^{\,i} : (x : A)\Delta}\ \textsc{TeleCons}$$

Figure 2.7: Theta Typing: Datatypes and Contexts

level. In the latter case, we add two declarations to the context. First, we add an abstract version of the datatype being defined (since the type constructor should be available when checking the types of the term constructors, but the term constructors themselves should not). Second, we add the parameters to the type constructor—these can be thought of as implicit arguments to each term constructor. Note that all arguments to type and term constructors are required to have mobile types. This is the reason datatypes are mobile themselves. The CData rule also ensures that recursive uses of the type constructor are strictly positive, a standard requirement which is necessary to ensure the consistency of the logical fragment. We omit the technical definition of strictly positive, but it can be found in the literature [40].

Our representation of dependent datatypes is somewhat unusual in that it does not explicitly include "indices"—arguments to the type constructor that may vary in the result type of each data constructor. We have chosen this representation for simplicity. It is still possible to encode invariants for which indices are typically used. Rather than having a constructor's return type instantiate a parameter, one may add an extra argument to the constructor assuming that the relevant parameter is equal to some other term. For example, to encode a length-indexed vector type with a natural-number parameter x, the nil constructor would take an extra argument of type $x = 0$, and the cons constructor would take two extra arguments: the length y of the tail and a proof that $x = \mathsf{Succ}\ y$. This example will be considered in greater detail in Section 2.3.1.

Term and type constructors are looked up from the context with rules TDCon, TTCon and TATCon. For type constructors, rules TTCon and TATCon simply look up the constructor in the context and ensure that the arguments to which it is applied correspond to its declared telescope of parameters, using the auxiliary judgement $\Gamma \vdash^{\theta} \overline{a_i}^{\,i} : \Delta$.

For term constructors, the situation is slightly more complicated. In most dependently typed languages, if d is a constructor of type $D\,\Delta$, the parameters Δ are arguments to d as well, since they appear in d's return type. For example, in Coq and Agda, the type $\mathsf{List}\,A$ has constructors nil and cons which take the type A of data contained in the list as an argument. However, rather than actually requiring d to be applied explicitly to an appropriate instantiation of these variables, the rule TDCon simply checks that a suitable instantiation of Δ exists. Of course, in practice, this is undecidable and these arguments must often be supplied explicitly in a surface language. But by leaving them out of the term constructor applications in the core language, we prevent type information from getting in the way of equalities.

Pattern matching expressions have the form:

$$\mathsf{case}\ a\ \mathsf{of}\ \{\ \overline{d_i\,\Delta_i \Rightarrow b_i}^{\,i \in 1 \ldots k}\ \}$$

Here we are using a telescope to represent the list of bound variables—this is the pattern being matched against. These expressions are checked by rule TCase. The first two premises check that the scrutinee a is a member of some datatype $D\,\overline{a_i}^{\,i}$

and that the match's return type B is well-formed. The third premise ensures that D is defined in the context. Note that we don't require any relationship between the universe level of B and the universe level of the datatype D. In particular, this permits *large eliminations*, which are types defined by pattern matching on values. Large eliminations are discussed in greater detail in Section 6.6.

The final premise checks the individual clauses of the pattern match. Since the parameters of the datatype are not arguments to each constructor, users do not match against them. Instead, when we extend the environment with the rest of the arguments to the appropriate constructor, we substitute the scrutinee's instantiation of D's parameters into the types of di's arguments. When checking a branch $d_i\,\Delta_i \Rightarrow b_i$, we also extend the context with a proof that $a = d_i\,\Delta_i$ (i.e., that the scrutinee is equal to the pattern being matched against).

Readers familiar with dependent pattern matching might be surprised that the final premise of this rule does not perform any substitutions in the return type of the match. Since indices are represented by adding equality hypotheses to the constructor types, these will be available to the type system for rewriting in each branch, along with the proof that the scrutinee is an application of the appropriate constructor. While this might be inconvenient in practice, we believe that a more standard pattern matching construct in the surface language could be compiled into our explicit version.

2.3 Examples

In this section we will show several example THETA programs, with the goal of highlighting how its type system solves the problems we described in the introduction. The design of a surface language for THETA and corresponding type inference algorithm are beyond the scope of this thesis. Therefore, we will show our examples as core THETA expressions instead, using a Coq and Agda-like concrete syntax. For readability, we will make use of a few surface language features that do not appear in THETA, but which are simple to translate into THETA expressions. In particular:

- We will make use of "implicit arguments" as they appear in Coq or Agda. This standard feature allows us to omit some arguments from function applications when the typechecker can easily fill in these arguments from the context. For example, as in the introduction, we write the type of vector append this way:

  ```
  app : {A : Type} → {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
  ```

  This indicates that we will omit the first three arguments when applying `app` because the typechecker can infer them from the types of the remaining arguments. In the corresponding core THETA program, these arguments always appear.

- In THETA, the domain type of every function and the type of each argument to a data constructor are required to be "mobile". In some cases, as in the type of `app` above, this requirement is satisfied naturally. In other cases, types must be tagged with a fragment using the @ type constructor to make them mobile. For example, because variables are not mobile, we might write the type of the polymorphic identity function this way:

```
(A : Type) → (A @ L) → A
```

This requirement can result in types that are harder to read than we would like, so the ZOMBIE surface language allows programmers to pick a "default" logicality by writing "`usually log`" or "`usually prog`" at the top of a file. It then automatically adds an appropriate @ to any type that is not mobile but needs to be. For readability, we will adopt the "`usually log`" convention in our examples—to obtain valid THETA programs, non-mobile function domains and data constructor argument types are implicitly tagged with `@ L`. Since the typing rules for introducing and eliminating @-types are not marked in the syntax, this does not require any change to the programs themselves.

- The `contra` term eliminates contradictory equalities between type constructors, but not term constructors. It is, however, possible to derive a similar elimination principle for contradictory equalities between term constructors *within* THETA. For example, we may eliminate equalities of the form `Zero = Succ n` using a large elimination:

  ```
  log f : Nat → Type
  f n = case n of
            Zero   → Nat
            Succ _ → Bool

  log nat_contra : (A : Type) → (n : Nat) → (eq : Zero = Succ n) → A
  nat_contra A n eq = contra
  ```

  We will explain how the use of `contra` in the definition of `nat_contra` typechecks in a few steps. First, observe that by TREFL, we know that `refl : f Zero = f Zero`. But we may use CONV along with the `eq` argument to `nat_contra` to turn this into a proof `f Zero = f (Succ n)`. Since `f Zero` reduces to `Nat` and `f (Succ n)` reduces to `Bool`, two more uses of TREFL and TCONV turn this into a proof of `Nat = Bool`, which can be eliminated by the built-in `contra`.

  None of this reasoning appears in the core THETA program above because the proof eliminated by `contra` does not appear in the syntax of the expression, but it would appear in the corresponding THETA typing derivation. Since `contra` for term constructors is derivable, it is supported directly in ZOMBIE, and we will use it our example programs below.

- In THETA, all recursive functions are written with the `rec` form, which is checked by two separate rules - one for terminating functions and one for unrestricted recursion in the programmatic fragment. In examples below, we will tag terminating recursion with the keyword `ind` and use unrestricted recursion in the programmatic fragment without explicitly mentioning `rec`. We will also write function definitions with multiple arguments, which are understood as nested single-argument lambdas and recursive functions.

- While THETA has an infinite hierarchy of universes, our examples require only `Type 0`, and we will write just `Type`.

- In the examples below, we will sometimes use the notation `let x = a in b`. Of course, THETA does not include let-bindings. However, they can be understood as β-expansions. That is, `let x = a in b` is equivalent to `(\x . b) a`.

  Sometimes, it is necessary to know inside the body `b` that the variable `x` is equal to the expression `a`. It is possible to achieve this with an additional β-expansion: `(\x . \y . b) a refl`, where the variable `y` here has the type `x = a`. None of our examples require us to reason explicitly with these equalities, so we will not introduce new syntax for this case.

All of the examples in this chapter have been implemented in ZOMBIE and may be found in the digital appendix available with this thesis [13]. In many cases, the ZOMBIE programs are substantially larger than the erased THETA programs shown here, because we must provide additional information about which equalities are necessary for typechecking. In these cases, we include a description of why the program typechecks to guide the reader's intuition about THETA equality.

We will focus on examples that exhibit the particular novel features of the THETA type system. Several additional examples have also been implemented in ZOMBIE and are available online [3]. For example, Vilhelm Sjöberg has implemented a SAT-solver and a unification algorithm for a simple tree language.

2.3.1 Example: Vector Append

In the introduction we used the example of proving that vector append is associative to illustrate some of the problems with how equality is represented in typical dependently typed languages. We will now revisit this example to examine how THETA handles it in detail.

We begin with natural numbers and `plus`

```
data Nat : Type where
  Zero
  Succ of (x : Nat)

log plus : Nat → Nat → Nat
ind plus n m =
  case n of
```

[3]`http://code.google.com/p/trellys/`

```
    Zero    → m
    Succ n' → Succ (plus n' ord m)
```

These definitions are standard except for the extra argument "**ord**" in the recursive call of `plus`. To understand this argument, recall the rule for terminating recursion in THETA:

$$\frac{\Gamma \vdash^{\mathsf{L}} (y : A) \to B : \mathsf{Type}_\ell \qquad \Gamma, y :^{\mathsf{L}} A, f :^{\mathsf{L}} (x : A) \to (z : x < y) \to [x/y]B \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} \mathsf{ind}\, f\ y.b : (y : A) \to B}\ \textsc{TInd}$$

When checking the body of an inductive function, the recursive call f (here `plus`) must be provided with the extra argument z—a proof that the expression on which f is called is a subterm of the original input. Here, in particular, we need to show that `n'` $<$ `n`. Such proofs are constructed by **ord**:

$$\frac{\Gamma \vdash^{\mathsf{L}} a : b = d_1\,\overline{b_i}^{\,i \in 1\ldots j}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ord} : b_i < b}\ \textsc{TOrd}$$

So we can see that for **ord** to typecheck here, it would be enough to have a proof of `n = Succ n'`. Happily, our rule for pattern matching adds exactly such an equality into the context when checking this branch (as the variable y in the final premise below):

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : D\,\overline{a_i}^{\,i} \quad \Gamma \vdash^{\mathsf{L}} B : \mathsf{Type}_{\ell_2} \\ \mathsf{data}\ D\,\Delta : \mathsf{Type}_{\ell_1}\ \mathsf{where}\ \{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i \in 1\ldots k}\,\} \in \Gamma \\ \forall i \in 1\ldots k,\quad \Gamma, [\overline{a_i}^{\,i}/\Delta]\Delta_i\,\theta, y :^{\mathsf{L}} a = d_i\,\Delta_i \vdash^{\theta} b_i : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{case}\ a\ \mathsf{of}\ \{\,\overline{d_i\,\Delta_i \Rightarrow b_i}^{\,i \in 1\ldots k}\,\} : B}\ \textsc{TCase}$$

In the concrete syntax of ZOMBIE, this reasoning is made more explicit by naming the equality proof provided by pattern matching and supplying it as an argument to **ord**. But ZOMBIE terms erase to core THETA terms before evaluation, so we obtain the program here that closely resembles the typical addition function.

With natural numbers in hand, we are prepared to represent vectors. As mentioned in Section 2.2.4, core THETA datatypes have parameters but no indices. Typically the length of a vector is represented with an index, so our definition of vectors is slightly unusual:

```
data Vec (A : Type) (n : Nat) : Type where
  Nil  of {n = 0}
  Cons of {sz : Nat} {n = Succ sz} A (Vec A sz)
```

Here we simulate indices by adding extra arguments to the constructors `Nil` and `Cons`. In the former case, we demand a proof that the length is zero, and in the latter case we demand a proof that the length is one greater than the length of the tail.

Carrying around these proofs at run-time would be inefficient, so ZOMBIE includes another feature: ICC*-style "compile-time only" annotations [7, 43]. This allows the user to tag these arguments as irrelevant at run time. We have not modeled this feature in THETA, so we will keep the proofs as real arguments but mark them as implicit since they are typically uninteresting and inferable.

We can define append for vectors inductively:

```
log app : {A : Type} → {n m : Nat}
        → Vec A n → Vec A m → Vec A (plus n m)
ind app v1 v2 =
  case v1 of
    Nil        → v2
    Cons a v1' → Cons a (app v1' ord v2)
```

There is quite a bit of implicit equality reasoning needed to typecheck this program. Consider the `Nil` branch of the pattern match. We are expected to return a term of type `Vec A (plus n m)`, but have supplied `vs` whose type is `Vec A m`. To see why this typechecks, recall from our definition of vectors that `Nil` actually carries with it a proof that the vector's length is zero, so when checking this clause of the match we have a proof of `n = 0` in the context. Additionally, the rule TREFL may be used to obtain a proof of `plus 0 m = m`:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{L}} a_1 = a_2 : \mathsf{Type}_\ell \\ a_1 \Rrightarrow^* b \quad a_2 \Rrightarrow^* b\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a_1 = a_2}\ \textsc{TRefl}$$

We can put these proofs together using rule TCONV to give `v2` the desired type.

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : [b_1/x]A \\ \Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \\ \Gamma \vdash^{\mathsf{L}} [b_2/x]A : \mathsf{Type}_\ell\end{array}}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \textsc{TConv}$$

In particular, one use of TCONV changes the type (`Vec A m`) to (`Vec A (plus 0 m)`) using the proof from `refl`, and a second use of TCONV changes the type (`Vec A (plus 0 m)`) to (`Vec A (plus n m)`) using the proof from `Nil`. In the current version of the source language ZOMBIE, the typechecker is able to infer both uses of conversion if the user provides the hint that `plus 0 m` should be reduced. Regardless of how much annotation is required in the source, however, the program is erased to the version shown here.

We would like to prove that vector append is associative, as described in the introduction. As we observed there, THETA's heterogeneous equality allows us to state and prove this theorem directly:

```
log app_assoc : {A : Type} → {n m k : Nat}
              → (v1 : Vec A n) → (v2 : Vec A m) → (v3 : Vec A k)
              → app v1 (app v2 v3) = app (app v1 v2) v3
```

```
ind app_assoc v1 v2 v3 =
  case v1 of
    Nil        → refl
    Cons a v1' → refl
```

Although typechecking this proof requires substantial equality reasoning, including a recursive call to `app_assoc`, we can see that none of this is reflected in the core THETA term itself because conversion is unmarked. To demonstrate how conversion is being used here, we will walk through how the `Cons` case of this program typechecks in detail.

Begin by observing that, by TREFL, we have:

```
refl : app v1 (app v2 v3) = app v1 (app v2 v3)
```

We will use this proof as a starting point and convert it to the desired type via multiple uses of TCONV. Recall that THETA's pattern matching rule will provide us with a proof of `v1 = Cons a v1'`, so we may use conversion to obtain:

```
refl : app v1 (app v2 v3) = app (Cons a v1') (app v2 v3)
```

We would like to unfold the definition of `app` on the right hand side of this equality. Since THETA is a call-by-value language, this expression is stuck and TREFL will not reduce it. However, we can still achieve the desired result via lambdas. First, observe that:

```
\x . refl : (x : Vec A k) → app (Cons a v1') x = Cons (a (app v1' x))
```

This equality typechecks where the other did not because x is a value and thus TREFL can reduce the use of `app`. We may apply this function to obtain:

```
(\x.refl) (app v2 v3)
    : app (Cons a v1') (app v2 v3) = Cons a (app v1' (app v2 v3))
```

Using this equality and TCONV with our previous equality, we obtain:

```
refl : app v1 (app v2 v3) = Cons a (app v1' (app v2 v3))
```

At this point, we would like to make a recursive call to get an equality involving the tail of the right-hand side. In particular:

```
app_assoc v1' ord v2 v3 : app v1' (app v2 v3) = app (app v1' v2) v3
```

Note that, as we saw before, the recursive call must be supplied with a proof **`ord`** that the new argument is smaller than the original input. Specifically, here **`ord`** must have the type `v1'` $<$ `v1`. But pattern matching provides a proof of `v1 = Cons a v1'`, so TORD can give **`ord`** the desired type.

Using the result of the recursive call and TCONV with the previous equality, we obtain:

```
refl : app v1 (app v2 v3) = Cons a (app (app v1' v2) v3)
```

Now we would like to fold the definition of `app` back up in order to obtain the desired result. We may use the same eta-expansion trick we used for the previous step of reduction to prove:

```
Cons a (app (app v1' v2) v3) = app (app (Cons a v1') v2) v3
```

Using this equality and TConv with the previous equality, we obtain:

```
refl : app v1 (app v2 v3) = app (app (Cons a v1') v2) v3
```

And since pattern matching provided a proof of `v1 = Cons a v1'`, a final use of TConv brings us to the desired equality:

```
refl : app v1 (app v2 v3) = app (app v1 v2) v3
```

It is clear that choosing not to mark uses of conversion in the syntax of expressions is buying us substantially simpler core programs. In the description of how just the `Cons` branch of the `app_assoc` theorem typechecks we used TConv five times. In a language like Coq, most of these require an explicit appeal to an equality eliminator in the proof itself.

In the remaining examples we will not describe how the programs typecheck in such detail. The curious reader may refer to the annotated versions that were typechecked with Zombie, where some of this reasoning is made explicit.

2.3.2 Example: Comparison and Course-of-Values Induction

The built-in terminating recursion operator in Theta supports only recursing on structural subterms. In the examples below, we will need "strong" or "course-of-values" induction for natural numbers, so we introduce it here.[4] The examples in this section include moderately involved mathematical reasoning, so readers interested in more practical examples are encouraged to skip to the next section and refer back to the definition of `LT` and the type given to course-of-values induction as necessary.

We begin by defining a "less than" relation for natural numbers. As before, we use equality arguments to simulate indices.

```
data LT (n : Nat) (m : Nat) : Type where
  LSucc of (m = Succ n)
  LStep of (m':Nat) (m = Succ m') (LT n m')
```

We will need several definitions and properties of `LT` before we can define course-of-values induction. We begin with a definition of predecessor for natural numbers and a simple related fact: while it is not the case that `n = Succ (pred n)` in general (because `n` may be 0), this is the case if there is anything less than `n`.

```
log pred : Nat → Nat
pred n = case n of
```

[4]The definition of course-of-values induction shown here is based on a Zombie version implemented by Nathan Collins.

```
              Zero    → Zero
              Succ n' → n'

log n_eq_SPn : (m n : Nat) → LT m n → n = Succ (pred n)
n_eq_SPn m n lt_m_n =
  case n of
    Zero →
      case lt_m_n of
        LSucc eq_0_Sn → contra
        LStep _ eq_0_Sn' _ → contra
    Succ n' → refl
```

The proof of `n_eq_SPn` is unsurprising. If `n` is `Succ n'`, then the result follows directly by reduction. If `n` is 0, we find a contradiction of the form `0 = Succ m'` by examining the proof of `LT m n`.

We also need to observe that `LT` enjoys a strong transitivity property: if `LT n m` and `LT m k`, then `LT n (pred k)`.

```
log lt_trans_pred : (n m k : Nat) → LT n m → LT m k → LT n (pred k)
ind lt_trans_pred n m k lt_n_m lt_m_k =
  case lt_m_k of
    LSucc eq_k_Sm → lt_n_m
    LStep k' eq_k_Sk' lt_m_k' →
      let ih = lt_trans_pred n m k' lt_n_m lt_m_k' ord in
       -- ih : LT n (pred k')
      LStep (pred k') (n_eq_SPn m k' lt_m_k') ih
```

Typechecking the above definition requires many (unmarked!) uses of TConv and TRefl. In the first case, we know that `k = Succ m`, and thus `pred k = m`. So, the proof that `LT n m` is also a proof of `LT n (pred k)`. In the second case, we know that `k = Succ k'` for some `k'`. A recursive call therefore yields that `LT n (pred k')`, and the result follows by `LStep`.

The implementation of course-of-values recursion is somewhat complicated, so we will first illustrate the technique used in a more specific setting. Consider the following ceiling division function for natural numbers:

```
log minus : Nat → Nat → Nat
ind minus n m =
  case m of
    Zero    → n
    Succ m' → case n of
                 Zero    → Zero
                 Succ n' → minus n' ord m'

prog div : Nat → Nat → Nat
div n m =
  case n of
```

```
    Zero    → Zero
    Succ n' → Succ (div (minus n' (pred m)) m)
```

We can see that, unlike the definition of `div` in the introduction, this implementation of division is safe because it always recurses on a strictly smaller number. However, because the recursion is not on a structural subterm, it cannot be be captured directly by THETA's built-in terminating recursor.

The trick we will use is to add a "dummy" argument to `div`. This dummy argument will descend by one in each recursive call, making the recursion structural. However, the function will still compute the correct division. To make this work, we will maintain the invariant that the "real" argument is less than the dummy argument.

```
log div' : (i n m : Nat) → LT n i → Nat
ind div' i n m n_lt_i =
  case n of
    Zero    → Zero
    Succ n' → Succ (div (pred i) ord (minus n' (pred m)) m
                        (lt_trans_pred ... n_lt_i))
```

Intuitively, `i` serves as a descending upper bound on the actual recursive argument. Typechecking `ord` in this recursive call requires somewhat more work than usual: since `LT n i`, we know that `i` is `Succ i'` for some `i'`. Since `i' = pred i`, we can show `pred i < i`. Additionally, the "..." in this example must be filled in by a proof that `LT (minus n' (pred m)) (Succ n')` (this is easily obtained but uninteresting).

With `div'` in hand, we can define `div` by picking `(Succ n)` for `i`:

```
log div : Nat → Nat → Nat
div n m = div' (Succ n) n m (LSucc refl)
```

We will now define a general course-of-values recursor. The type we assign to `cov` is unsurprising: given a property `p` of natural numbers, `cov` constructs a proof that `p` holds for any natural number, provided we have a proof that, if `p m` holds for all `m` that are less than `n`, then `p n` holds as well.

```
log cov : (p:Nat → Type)
       → ((n:Nat) → ((m:Nat) → LT m n → p m) → p n)
       → (n:Nat) → p n
cov p f =
  -- g : (i:Nat) -> (n:Nat) -> (LT n i)
  --   -> ((m:Nat) -> (LT m n) -> p m))
  let g  =
   ind g i = \n n_lt_i m m_lt_n .
     let m_lt_i' = lt_trans_pred m n i m_lt_n n_lt_i in
      -- m_lt_i' : m < pred i
     f m (g (pred i) ord m m_lt_i')
  in \n . f n (g (Succ n) n (LSucc refl))
```

The implementation of course-of-values induction is somewhat complex. Intuitively, however, it uses the same "dummy argument" trick as in our implementation of `div` above. In particular, `cov` takes in a "recursive template" `f`, and then implements a structurally recursive version of this template `g` by adding a dummy argument.

2.3.3 Example: Merge Sort

In this section we will examine merge sort, a classic example for dependently-typed languages [3]. Merge sort is an excellent example for THETA because, while it is total, it is not structurally recursive. The basic idea of merge sort is very simple: take a list, split it in half, sort the halves, merge them back together. Unfortunately we get the two halves of a list by calling another function to split it, and thus the recursion on these two halves is not structural, even though it is clear they are both smaller.

There are several approaches to getting around this problem in languages that demand structural recursion. One option is to index merge sort by the length of the list and observe that this length will go down in the recursive calls. Another option is to use an intermediate binary tree data structure so that the recursion become structural. In either case, we have modified the behavior and the type of this standard algorithm, which is somewhat unsatisfactory. The termination argument for merge sort is "polluting" its definition.

With THETA, we have a new option. We will implement merge sort in the programmatic fragment where we are allowed to use "unsafe" general recursion. Thus, its definition will look just like we had implemented it in a standard functional language like Haskell or OCaml. After we have defined merge sort, we will give a separate proof that it always terminates. This verifies *extrinsically* the termination property that must be encoded *intrinsically* in a language like Coq or Agda. It will also allow us to use merge sort in the logical fragment.

The definition of merge sort is relatively standard. We begin with definitions of a few datatypes. Lists are the things we'll sort. Pairs are needed for the return type of the function that splits a list into halves. Booleans are needed for the type of a comparison function to use while sorting. We will use if/then/else notation where convenient rather than explicitly pattern matching on booleans.

```
data Prod (A:Type) (B:Type) : Type where
  Pair of (x:A) (y:B)

data Bool : Type 0 where
  True
  False

data List (A:Type) : Type where
  Nil
  Cons of (a:A) (xs:List A)
```

Merge sort itself comprises three functions: one to divide lists in half, one to merge sorted lists together, and the top-level sort function itself. We define all three functions in the programmatic fragment, although `split` and `merge` could be defined logically without much effort, using `ordtrans` in the former case to handle the deep pattern matching, and using a nested recursion in the latter case to handle recursing on two lists.

We begin with `split`, which takes in a list and splits it into two lists. It is implemented by pattern matching on the input list. In the case where it has at least two elements, we recurse on the tail and add one of the first two elements to each of the resulting lists.

```
prog split : {A : Type} → List A → Prod (List A) (List A)
split xs =
  case xs of
    Nil → Pair Nil Nil
    Cons x1 Nil → Pair (Cons x1 Nil) Nil
    Cons x1 (Cons x2 xs') →
      case (split xs') of
        Pair xs1 xs2 → Pair (Cons x1 xs1) (Cons x2 xs2)
```

Next we define `merge`, which merges two sorted lists. It is implemented by pattern matching on the lists. In the case where both contain at least one element, the smaller of the two becomes the head of the resulting list. For convenience, we pattern match on two variables at once—this is not technically allowed in THETA, but can be easily expanded into a nested match. Both `merge` and `sort` take a comparison function for the elements of the list as an argument.

```
prog merge : {A:Type} → (A → A → Bool) → List A → List A → List A
merge lt xs ys =
   case xs , ys of
     Nil , _ → ys
     _ , Nil → xs
     Cons x xs' , Cons y ys' →
       if (lt x y) then Cons x (merge lt xs' ys)
                   else Cons y (merge lt xs ys')
```

Finally we have `sort` itself.

```
prog sort : {A : Type} → (A → A → Bool) → List A → List A
sort lt xs =
  case xs of
    Nil → Nil
    Cons x Nil → Cons x Nil
    _ →
      case (split xs) of
        Pair xs1 xs2 → merge lt (sort lt xs1) (sort lt xs2)
```

We would like to prove that `sort` always terminates. The first task is to state this in the language itself. For this purpose, we need existential types:

```
data Sigma (A : Type) (B : (x:A) → Type) : Type where
  Ex of (x : A) (pf : B x)
```

As a matter of notational convenience, we will write `Sigma (x:A) B` rather than `Sigma A (\x . B)` in the code below. We may assert that an expression terminates using an existential:

```
log sort_terminates : {A : Type} → (lt : A → A → Bool) → (xs : List A)
                    → Sigma (ys : List A) (ys = sort lt xs)
```

Since this function is in the logical fragment, it is guaranteed to terminate. When it does, we will have a list value and a proof that this value is equal to `sort lt xs`. Thus, because the language is confluent, `sort` must terminate on these arguments.

To show that `sort` terminates, we must show that the two functions it calls also terminate. We begin with `merge`. Note that we define `merge_terminates` using a nested induction, since when `merge` recurses one of its two input lists will get smaller and we must have an induction hypothesis available for either case.

```
log merge_terminates : {A:Type} → (lt : A → A → Bool)
                     → (xs1 xs2 : List A)
                     → Sigma (ys : List A) (ys = merge lt xs1 xs2)
ind merge_terminates lt xs1 =
  ind mt' xs2 =
    case xs1 , xs2 of
      Nil ,          Nil          → Ex Nil refl
      Nil ,          Cons x2 xs2' → Ex xs2 refl
      Cons x1 xs1' , Nil          → Ex xs1 refl
      Cons x1 xs1' , Cons x2 xs2' →
        case (lt x1 x2) of
          True →
            case (merge_terminates lt xs1' ord xs) of
              Ex ys ys_eq → Ex (Cons x1 ys) refl
          False →
            case (mt' xs2' ord) of
              Ex ys ys_eq → Ex (Cons x2 ys) refl
```

There are two parts to understanding this proof. The first is understanding how the witness of termination is constructed (i.e., the first element of the result pair). When one of the lists is empty, `merge` simply returns the other, so constructing the witness list is quite easy in the first three cases here. Constructing the witness when both lists are non-empty is slightly more complicated. We begin by checking which head element is smaller, since this is the head of the new list. Then we call one of the two induction hypotheses (`merge_terminates` itself, or `mt'`) depending on which list gets smaller. This provides a witness of the termination of `merge`'s recursive call, which is the tail of the new list.

While the core THETA term shown here constructs the witness of termination quite explicitly, it does not show the equational reasoning necessary to prove that this witness is the result of merge. This is unsurprising, since such reasoning will be used by TCONV to give `refl` the appropriate type, and uses of TCONV are unmarked in the syntax. Spelling out this reasoning in detail would take a number of (likely uninformative) pages, so we will not consider it in great detail. At a high level, the underlying THETA derivation observes that the structure exposed by pattern matching on the lists allows `merge` to unfold once, and then uses the equality from the appropriate induction hypothesis to close the gap. Much of this reasoning is made more explicit in the source language version this function.

It remains to show that `split` terminates and then that `sort` itself terminates. Since `sort` is not structurally recursive, the built-in `ind` recursor we used in the proof of `merge_terminates` will be insufficient. Instead we will use course-of-values recursion on the length of the list, as defined in the previous section. This will require us to show that the lists on which `sort` calls itself have a smaller length than the input list. Since these lists are themselves the result of a call to `split`, our proof of `split_terminates` will need to provide more than a simple termination witness—it will also need to produce proofs that the result lists are smaller than the input list.

For this purpose, we define a length operation on lists.

```
log length : {A : Type} → List A → Nat
ind length xs =
  case xs of
    Nil        → Zero
    Cons _ xs' → Succ (length xs' ord)
```

The proof that `split` terminates will also require a few simple facts about natural numbers and the less-than relation. We omit the proofs of these lemmas, as they are uninteresting:

```
log lt_1_0_contra : {A : Type} → LT (Succ Zero) Zero → A

log lt_1_1_contra : {A : Type} → LT 1 1 → A

log LT_SS : {n m : Nat} → LT n m → LT (Succ n) (Succ m)

log LT_1_SSm : (m : Nat) → LT 1 (Succ (Succ m))
```

We are now prepared to prove that `split` terminates. As described above, we actually prove a more expressive theorem that characterizes the sizes of the result lists. In particular, we show that if the input list had at least two elements, then the result lists have a smaller length. This is the most complicated part of the termination proof for `sort`.

```
log split_terminates : {A : Type} → (xs : List A)
      → Sigma (ys1 : List A) (Sigma (ys2 : List A)
            (Prod (split A xs = Pair ys1 ys2)
                  (LT 1 (length xs)
                     → Prod (LT (length ys1) (length xs))
                            (LT (length ys2) (length xs))))))
ind split_terminates xs =
case xs of
  Nil → Ex Nil (Ex Nil (Pair refl (\lt_1_0 . lt_1_0_contra lt_1_0)))
  Cons a Nil →
    Ex (Cons a Nil)
       (Ex Nil (Pair refl (\lt_1_1 . lt_1_1_contra lt_1_1)))
  Cons a (Cons a' xs'') →
    case (split_terminates xs'' ord) of
      Ex ys1 (Ex ys2 (Pair split_eq flen)) →

         -- flen' : LT 1 (length xs)
         --      -> Prod (LT (length ys1) (length xs))
         --              (LT (length ys2) (length xs))
        let flen' = \lt_1__len_xs .
          case (length xs'') of
            Zero → Pair (LSucc refl) (LSucc refl)
            Succ Zero →
              Pair (LSucc refl) (LStep 2 refl (LSucc refl))
            Succ (Succ n') →
               -- lt_1_lxs'' : LT 1 (length xs'')
              let lt_1_lxs'' = LT_1_SSm (pred (pred (length xs'')))
               in
              case (flen lt_1_lxs'') of
                Pair lt__lys1_lxs'' lt__lys2_lxs'' →
                  Pair (LStep (Succ (length xs'')) refl
                              (LT_SS lt__lys1_lxs''))
                       (LStep (Succ (length xs'')) refl
                              (LT_SS lt__lys2_lxs''))
          in
         Ex (Cons a ys1) (Ex (Cons a' ys2) (Pair refl flen'))
```

Predictably, the portion of this function which computes the witnesses to termination is relatively straightforward. The more complicated reasoning in this term is used to construct the proof about the length of the result lists. In a language like Coq, this proof would be more conveniently constructed via tactics, but here we build it explicitly.

In particular, in each case we must construct a function of type:

```
LT 1 (length xs) → Prod (LT (length ys1) (length xs))
                        (LT (length ys2) (length xs))
```

Here, `xs` is the input list and `ys1` and `ys2` are the output lists. In the cases where `xs` has zero or one elements, this is simple. The input inequality is a contradiction, and we use two of the lemmas mentioned above to eliminate it.

In the case where `xs` has the form `Cons a (Cons a' xs'')`, we know that `ys1` and `ys2` have the forms `Cons a ys1'` and `Cons a' ys2'`, respectively. Here `ys1'` and `ys2'` are the results of `split xs''`, provided by the induction hypothesis of our lemma.

Thus, it will be enough to show, along with a similar result for `ys2’`, that:

```
  LT (Succ (length ys1’)) (Succ (Succ (length xs’’)))
```

Happily, the induction hypothesis relates the length of `ys1’` and the length of `xs’’` for us, when the latter has at least two elements. And in the case where `xs’’` has zero or one elements, we know the exact lengths of all the lists involved.

We can now prove that `sort` terminates. The proof goes by course-of-value induction on the length of the list being sorted.

```
log sort_terminates : {A:Type} → (lt : A → A → Bool)
                    → (xs : List A)
                    → Sigma (ys : List A) (ys = sort lt xs)
sort_terminates lt ys =
  let st_def =
    (\n st_rec xs xs_len_eq .
       case xs of
         Nil        → Ex Nil refl
         Cons a Nil → Ex (Cons a Nil) refl
         Cons a (Cons a’ xs’’) →
           let split_term = split_terminates xs in
           case split_term of
             Ex ys1 (Ex ys2 (Pair ys_eq ys_length)) →
               let len_LT = LT_1_SSm (length xs’’) in
                -- len_LT : LT 1 (length xs)
               case (ys_length len_LT) of
                 Pair ys1_length ys2_length →
                   let ih1 = st_rec (length ys1) ys1_length ys1 refl in
                   let ih2 = st_rec (length ys2) ys2_length ys2 refl in
                   case ih1 , ih2 of
                     Ex ys1_sorted _ , Ex ys2_sorted _ →
                       merge_terminates lt ys1_sorted ys2_sorted)
  let st = cov (\n . (xs: List A) → (n = length xs)
                        → Exists (ys : List A) (ys = sort lt xs))
               st_def (length ys) in
   -- st : (xs : List A) -> (length ys = length xs)
   --                      -> Sigma (ys : List A) (ys = sort lt xs))
  st ys refl
```

The bulk of the definition is in the intermediate function `st_def`. This is the “recursive template” passed to our course-of-values recursor `cov`. Intuitively, it takes as an argument a proof that `sort` terminates for any list whose length is less than `n` (`st_rec`), and it proves that `sort` terminates for any list of length `n`. Its type is:

```
  st_def : (n:Nat)
         → ((m:Nat) → LT m n → (xs:List A) → m = length xs
                    → Sigma (ys : List A) (ys = sort lt xs))
         → (xs:List A) → n = length xs
         → Sigma (ys : List A) (ys = sort lt xs)
```

Its implementation is relatively straightforward. When the input list has only zero or one elements, it constructs the sorted list directly. For longer lists, `sort` calls `split` and then `merge`. So `st_def` first calls `split_terminates` to obtain a proof that the call to `split` returns lists `ys1` and `ys2` whose length is less than the length of the original list. This fact about their lengths allows two "recursive" calls to `st_def` to produce the sorted versions of `ys1` and `ys2`. The definition of `sort` simply merges these sorted lists, so use `merge_terminates` to conclude.

Having proved that `sort` terminates, we may obtain a logical version of it and prove that the logical version agrees with the programmatic version:

```
log lsort : {A : Type} → (A → A → Bool) → List A → List A
lsort lt xs =
  case (sort_terminates lt xs) of
    Ex xs' _ → xs'

log lsort_eq_sort : {A : Type} → (lt : A → A → Bool) → (xs :List A)
                  → (lsort lt xs = sort lt xs)
lsort_eq_sort lt xs =
  case (sort_terminates lt xs) of
    Ex xs' xs'_eq_sort_xs → xs'_eq_sort_xs
```

2.4 Conclusion

In this chapter we exhibited the design of Theta, a new dependently typed core language. Theta can define and reason about potentially non-terminating functions, and it includes a notion of equality that is not constrained by types and whose uses do not pollute terms.

We then showed, by example, that Theta's novel features solve the problems with dependently typed programming described in the introduction. We began by revisiting the proof of the associativity for vector append, a property that is difficult even to state elegantly in Coq and Agda. In Theta, we were able to give it the expected type and a simple definition. Then, after a detour through course-of-values induction, we considered the classic example of merge sort. Even though merge sort is not structurally recursive, it may be defined directly in Theta's programmatic fragment. This fragment provides no termination guarantee for merge sort, but we showed it is possible to prove that the algorithm terminates after the fact. Thus, the termination argument need not clutter the definition of merge sort itself.

The proof that merge sort terminates is somewhat complicated, and because of the untyped nature of Theta terms it can be difficult to follow. However, we do not believe this to be a substantial problem for Theta. The proofs shown here resemble Coq proof terms—with appropriate tool support, like a tactic language, it would not be necessary to construct them directly.

THETA is an extension of PCC$^\theta$ with several features that are useful for examples. In the next few chapters, we will examine the metatheory of PCC$^\theta$ to gain confidence that our design is reasonable. While the theory of THETA has proven difficult (as described in detail in Chapter 6), we have shown type safety and consistency (of the logical fragment) for several sublanguages of THETA, including PCC$^\theta$. These proofs have even been mechanized for a sublanguage with dependent types. Thus, we are optimistic that the design shown here is a sensible solution to some of the problems with current dependently typed languages.

Chapter 3

Partially Step-Indexed Logical Relations for Normalization

You may not like Tate's methods,
But you've got to admit,
She's a real nice kid.

Linda Blair was Born Innocent
The Mountain Goats

In the previous chapter, we described a language with two novel features: it allowed general recursion by splitting the language into two fragments, and it featured a notion of equality whose uses were completely unmarked in the syntax of terms. Both features substantially complicate the metatheory of the language, and they do it in largely orthogonal ways.

This chapter explores the complications introduced by adding general recursion in the context of a simply typed language. This language is called λ^θ. The language λ^θ can be thought of as a drastically cut down version of Theta. It features a programmatic fragment with non-termination and a logical fragment which is more restricted. The primary technical contribution of this chapter is a proof of normalization for the logical fragment of λ^θ. This proof makes use of a new technique, called *partially step-indexed logical relations*. A language similar to λ^θ and the technique we exhibit in this chapter first appeared in previous work with Vilhelm Sjöberg and Stephanie Weirich [15].

We begin with λ^θ's definition (Section 3.1) and a demonstration that it enjoys a standard progress property (Section 3.2.1). Since the language is simply typed, it features no notion of provable equality—that feature will be further explored in Chapter 4. However, in keeping with our overall goals, λ^θ records very little typing information in the terms. For example, uses of the "@" type constructor are unmarked. This introduces minor difficulties which foreshadow the development of the next two chapters, especially in the inversion lemmas needed for the proof of preservation (Section 3.2.2).

Types

$A, B \quad ::= \quad \mathsf{Unit} \mid A \to B \mid A + B \mid A@\theta \mid \alpha \mid \mu\,\alpha.A$

Terms

$a, b \quad ::= \quad x \mid \mathsf{rec}\,f\;x.a \mid a\;b \mid () \mid \mathsf{roll}\;a \mid \mathsf{unroll}\;a$
$\qquad\qquad \mid \mathsf{inl}\;a \mid \mathsf{inr}\;a \mid \mathsf{case}\;\;a\;\;\mathsf{of}\;\{\mathsf{inl}\;x \Rightarrow a_1; \mathsf{inr}\;x \Rightarrow a_2\}$

Consistency Classifiers

$\theta \quad ::= \quad \mathsf{L} \mid \mathsf{P}$

Environments

$\Gamma \quad ::= \quad \cdot \mid \Gamma, x :^{\theta} A$

Values

$v \quad ::= \quad x \mid () \mid \mathsf{inl}\;v \mid \mathsf{inr}\;v \mid \mathsf{rec}\,f\;x.a \mid \mathsf{roll}\;v$

Syntactic Abbreviation:

$$\lambda x.a \quad \triangleq \quad \mathsf{rec}\,f\;x.a \qquad \text{when } f \notin \mathrm{FV}(a)$$

Figure 3.1: λ^{θ}: Syntax

To motivate the "partially step-indexed" technique, we will demonstrate that direct adaptations of the Girard–Tait reducibility method [27, 58] are insufficient to prove normalization of λ^{θ}'s logical fragment (Section 3.3). Since logical terms are permitted to make use of certain values computed programmatically, it is necessary to simultaneously verify partial correctness properties of the programmatic fragment. Thus, the new technique combines the traditional method with step-indexed logical relations [2, 5]. The chapter concludes with an introduction of this technique (Section 3.3.3) and a demonstration that it is sufficient to prove λ^{θ}'s logical fragment normalizes (Section 3.3.4).

The metatheoretic results in this chapter have been completely mechanized in the Coq proof assistant, and are available in the `LTheta` subdirectory of this thesis's digital appendix [13]. Since there is no doubt about the correctness of the results, we avoid monotonous details and focus on clarity in explaining the techniques used and how our proofs differ from those for the simply typed lambda calculus without a programmatic fragment. Proofs for a larger system are written out in detail in Chapter 5.

3.1 Language Definition

The language that we consider in this chapter is a variant of the simply typed call-by-value lambda calculus with recursive types and general recursion. Its syntax is given in Figure 3.1. The chief novelty is the presence of *consistency classifiers* θ. These classifiers are used by the typing judgement (written $\Gamma \vdash^{\theta} a : A$) to divide the language into two fragments. The logical fragment, denoted by L, is a simply typed lambda calculus with unit and sums. As we will show, all terms in this fragment are normalizing. The programmatic fragment, denoted by P, adds general recursion and recursive types. The programmatic fragment is a strict superset of the logical fragment: if $\Gamma \vdash^{\mathsf{L}} a : A$, then $\Gamma \vdash^{\mathsf{P}} a : A$ as well.

Terms in the language may include subexpressions from both fragments. The $A@\theta$ type form mark such transitions. Intuitively, the judgement $\Gamma \vdash^{\theta} a : A@\theta'$ holds when fragment θ can safely observe that a has type A in the fragment θ'.

3.1.1 The Typing Judgement

We now describe the typing rules, given in Figure 3.2. As shown in rule TVar, variables in the typing context are tagged with a fragment. When a value is substituted for a variable, the value must check in the corresponding fragment.

There are two rules for type-checking functions. The first, TLam, checks non-recursive functions in the logical fragment. Here, $\lambda x.b$ is syntax sugar for $\mathsf{rec} f\, x.b$ when f does not occur free in b. The second rule, TRec, checks (potentially) recursive functions in the programmatic fragment. Observe that, in both cases, the argument is assumed to check in the same fragment as the function itself. However, it is still possible for a function in one fragment to take an argument from the other fragment using the domain type $A@\theta$, which we discuss next. Additionally, we require that the domain type of a function be "mobile" using the judgement $\mathsf{Mob}(A)$. This requirement is explained in detail when we describe mobile types below. The rule for function application, TApp, is standard.

The type form $A@\theta$ internalizes the typing judgement. Three rules assign @-types, describing the circumstances in which the fragments may safely talk about each other. The first rule, TBoxP, says that the programmatic fragment may internalize any typing judgement—if a has type A in fragment θ, then the programmatic fragment can observe that a has type $A@\theta$.

Rules TBoxL and TBoxLV internalize the typing judgement in the logical fragment and are restricted to ensure termination. The former says that if a itself has type A in the logical fragment, then a may also be given the type $A@\theta$ for any θ (since logical terms are also programmatic). The latter permits the logical fragment to observe that a term checks programmatically. In that case, the term must be a value to ensure normalization. This restriction still permits the logical fragment to consider programmatic terms (for example, recursive functions are values). The @-types are eliminated by the rule TUnboxVal, which is restricted to values to

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{x :^{\theta} A \in \Gamma}{\Gamma \vdash^{\theta} x : A}\ \textsc{TVar} \qquad \frac{\Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} b : B \quad \mathsf{Mob}(A)}{\Gamma \vdash^{\mathsf{L}} \lambda x.b : A \to B}\ \textsc{TLam} \qquad \frac{\Gamma \vdash^{\theta} b : A \to B \quad \Gamma \vdash^{\theta} a : A}{\Gamma \vdash^{\theta} b\ a : B}\ \textsc{TApp}$$

$$\frac{\Gamma, y :^{\mathsf{P}} A, f :^{\mathsf{P}} A \to B \vdash^{\mathsf{P}} a : B \quad \mathsf{Mob}(A)}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\ y.a : A \to B}\ \textsc{TRec} \qquad \frac{\Gamma \vdash^{\theta} a : A}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}\ \textsc{TBoxP} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{L}} a : A@\theta}\ \textsc{TBoxL}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\ \textsc{TBoxLV} \qquad \frac{\Gamma \vdash^{\theta} v : A@\theta'}{\Gamma \vdash^{\theta'} v : A}\ \textsc{TUnboxVal} \qquad \frac{}{\Gamma \vdash^{\mathsf{L}} () : \mathsf{Unit}}\ \textsc{TUnit}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\ \textsc{TSub} \qquad \frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \mathsf{Mob}(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\ \textsc{TMobVal} \qquad \frac{\Gamma \vdash^{\theta} a : A}{\Gamma \vdash^{\theta} \mathsf{inl}\, a : A + B}\ \textsc{TInl}$$

$$\frac{\Gamma \vdash^{\theta} b : B}{\Gamma \vdash^{\theta} \mathsf{inr}\, b : A + B}\ \textsc{TInr} \qquad \frac{\Gamma \vdash^{\theta} a : (A_1 + A_2)@\theta' \quad \Gamma, x :^{\theta'} A_1 \vdash^{\theta} b_1 : B \quad \Gamma, x :^{\theta'} A_2 \vdash^{\theta} b_2 : B}{\Gamma \vdash^{\theta} \mathsf{case}\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1; \mathsf{inr}\ x \Rightarrow b_2\} : B}\ \textsc{TCase}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} a : [\mu\,\alpha.A/\alpha]A}{\Gamma \vdash^{\mathsf{P}} \mathsf{roll}\, a : \mu\,\alpha.A}\ \textsc{TRoll} \qquad \frac{\Gamma \vdash^{\mathsf{P}} a : \mu\,\alpha.A}{\Gamma \vdash^{\mathsf{P}} \mathsf{unroll}\, a : [\mu\,\alpha.A/\alpha]A}\ \textsc{TUnroll}$$

$$\boxed{\mathsf{Mob}(A)}$$

$$\frac{}{\mathsf{Mob}(\mathsf{Unit})}\ \textsc{MUnit} \qquad \frac{\mathsf{Mob}(A) \quad \mathsf{Mob}(B)}{\mathsf{Mob}(A + B)}\ \textsc{MSum} \qquad \frac{}{\mathsf{Mob}(A@\theta)}\ \textsc{MAt}$$

Figure 3.2: λ^{θ}: Typing

preserve termination in the logical fragment.

Rules TUnit, TInl and TInr deal with the introduction forms for the unit and sum base types. These terms may be used in either fragment and the typing rules are standard. Rule TCase checks the pattern matching elimination form for sums. Notably, sums that typecheck in one fragment may be eliminated in the other. We use the @ type to ensure that this does not introduce non-termination into the logical fragment. To pattern match on a term a from fragment θ' in fragment θ, we require that $\Gamma \vdash^{\theta} a : (A_1 + A_2)@\theta'$. That is, we require that *fragment θ can safely observe that 'a' has a sum type in fragment θ'*. The restrictions built into the @-type introduction rule ensure that a terminates if θ is L. In Theta this was unnecessary because all arguments to datatypes were mobile, but we have relaxed that restriction in λ^{θ}.

Two rules describe the relationship between the fragments. As already discussed, any logical term can be used programmatically—this is the content of rule TSub. Rule TMobVal is more interesting. It allows potentially dangerous programmatic terms to be used in the logical fragment in certain circumstances. In particular, the term must be a value (to ensure termination) and its type must be "mobile". The mobile restriction, formalized by the $\mathsf{Mob}(A)$ judgement in the same figure, intuitively means that we move can move *data* but not *computations* from the programmatic fragment to the logical one. For example, moving a natural number computed in P to L is safe, but moving a function from P to L could cause non-termination when the function is applied.

Importantly, $A@\theta$ is a mobile type for any A. The programmatic fragment is permitted to compute logical values, including logical function values, and pass them back to the logical fragment. When the language is extended with dependent types, this becomes useful for working with proofs. For example, a partial decision procedure could be written in the programmatic fragment and the resulting proofs could be used in the logical fragment if the procedure terminates (as in the SAT solver example from Chapter 1).

We can now explain the requirement that the domain types of functions must be mobile. Recall that when checking a function, we assume it will be passed an argument that lives in the same fragment as the function. But suppose we define a function in the logical fragment and then move it to the programmatic fragment with rule TSub. Now the application rule will permit the function to be applied to programmatic arguments, but the body of the function might depend on the assumption that the argument will be logical.

The requirement that function domains are mobile eliminates this problem by ensuring functions are only ever applied to types whose values are the same in both fragments. Since any type may be made mobile by tagging it with a fragment, this does not restrict expressiveness. In earlier versions of this work, we solved this problem in a different way: by explicitly tagging all function domains with a fragment. The present solution improves on that approach by only requiring tags for functions whose domains aren't naturally mobile.

Evaluation contexts

$$\begin{array}{lll} E & ::= & [\cdot] \mid E\, b \mid v\, E \mid \mathsf{roll}\, E \mid \mathsf{unroll}\, E \mid \mathsf{inl}\, E \mid \mathsf{inr}\, E \\ & & \mid \mathsf{case}\ E\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \end{array}$$

$$\boxed{a \leadsto b}$$

$$\frac{}{(\mathsf{rec} f\ x.a)\ v \leadsto [v/x][\mathsf{rec} f\ x.a/f]a}\ \textsc{SBeta}$$

$$\frac{}{\mathsf{case\ inl}\, v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \leadsto [v/x]a_1}\ \textsc{SCaseL} \qquad \frac{}{\mathsf{unroll}\,(\mathsf{roll}\, v) \leadsto v}\ \textsc{SUnroll}$$

$$\frac{}{\mathsf{case\ inr}\, v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \leadsto [v/x]a_2}\ \textsc{SCaseR} \qquad \frac{a \leadsto b}{E[a] \leadsto E[b]}\ \textsc{SCtx}$$

$$\boxed{a \leadsto^n b} \qquad\qquad\qquad \boxed{a \leadsto^* b}$$

$$\frac{}{a \leadsto^0 a}\ \textsc{MSRefl} \qquad \frac{a \leadsto^k b \quad b \leadsto b'}{a \leadsto^{(k+1)} b'}\ \textsc{MSStep} \qquad \frac{a \leadsto^k b}{a \leadsto^* b}\ \textsc{ASAny}$$

Figure 3.3: λ^θ: Operational Semantics

Finally, the language includes iso-recursive types [50]. These are checked by the two rules TRoll and TUnroll. Recursive types with negative occurrences—that is, with the recursive variable appearing to the left of an arrow, such as $\mu\,\alpha.(\alpha \to A)$—are a potential source of nontermination. To ensure normalization for the logical fragment, we restrict recursive types to the programmatic fragment (we will revisit and relax this restriction in Chapter 4).

3.1.2 Operational Semantics

The language's operational semantics are given in Figure 3.3. We use standard call-by-value evaluation contexts and a small-step reduction relation. The multi-step reduction relation is indexed by a natural number—this will be useful in the step-indexed logical relation defined in Section 3.3.3.

3.2 Syntactic Metatheory

We begin our examination of λ^θ's metatheory with a syntactic proof of type safety via progress and preservation lemmas [65]. As mentioned above, we will focus on explaining the structure of our proofs and how they differ from those for a more traditional language.

3.2.1 Canonical Forms and Progress

Canonical forms lemmas are used to classify the possible closed values for each type. For example, we have the following lemma for sum types:

Lemma 3.2.1 (Canonical forms for sums)**.** If $\cdot \vdash^{\theta} v : B_1 + B_2$ then there is some v' such that $v = \mathsf{inl}\, v'$ or $v = \mathsf{inr}\, v'$.

This lemma is usually proved directly by induction on the given typing derivation. However, in our setting, this approach gets stuck in the case for the rule TUnboxVal. Instantiated for our lemma, it has the form:

$$\frac{\cdot \vdash^{\theta} v : (A + B)@\theta'}{\cdot \vdash^{\theta'} v : A + B}\ \textsc{TUnboxVal}$$

The problem is that the conclusion of this rule matches the theorem but the hypothesis does not, so we have no induction hypothesis to tell us about v. The solution is to generalize the statement of the lemma to account for the possibility of @ types:

Lemma 3.2.2 (Generalized canonical forms for sums)**.** Suppose $\cdot \vdash^{\theta} v : A$ and $A = B_1 + B_2$ or $A = (...((B_1 + B_2)@\theta_1)...)@\theta_n$ for some $\theta_1, ..., \theta_n$. Then there is some v' such that $v = \mathsf{inl}\, v'$ or $v = \mathsf{inr}\, v'$.

This generalized statement of the canonical forms lemma is provable directly by induction on the typing derivation. The original statement follows immediately as a corollary. We prove canonical forms lemmas for the other types using the same technique.

Lemma 3.2.3 (Canonical forms for Unit)**.** If $\cdot \vdash^{\theta} v : \mathsf{Unit}$ then $v = ()$.

Lemma 3.2.4 (Canonical forms for arrows)**.** If $\cdot \vdash^{\theta} v : A \to B$ then $v = \mathsf{rec} f\, x.b$ for some f, x and b.

Lemma 3.2.5 (Canonical forms for recursive types)**.** If $\cdot \vdash^{\theta} v : \mu\, \alpha.A$ then $v = \mathsf{roll}\, v'$ for some v'.

With these lemmas, we can prove a standard progress theorem. We show only the application case. The other cases are similarly straightforward.

Theorem 3.2.6 (Progress)**.** If $\cdot \vdash^{\theta} a : A$ then either a is a value or $a \leadsto a'$ for some a'.

Proof. By induction on the typing derivation $\mathcal{D} :: \cdot \vdash^{\theta} a : A$. Consider the application case:

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\cdot \vdash^{\theta} b : A \to B} \qquad \overset{\mathcal{D}_2}{\cdot \vdash^{\theta} a : A}}{\cdot \vdash^{\theta} b\, a : B}$ TApp

  Since $b\ a$ is not a value, we must show that it steps.

  The IHs for $\mathcal{D}_1$ and $\mathcal{D}_2$ give us that both b and a either step or are themselves values. If b steps, then $b\ a$ steps because $E\ a$ is an evaluation context. Similarly, if b is a value v but a steps, $b\ a$ steps because $v\,E$ is an evaluation context.

  So suppose both b and a are values. By canonical forms for arrow types (Lemma 3.2.4) and $\mathcal{D}_1$, we know that $b = \mathsf{rec}\, f\ x.b'$ for some f, x and b'. Thus, $b\ a = (\mathsf{rec}\, f\ x.b')\ a$ steps by rule SBeta, since a is a value.

□

3.2.2 Substitution, Inversion and Preservation

For preservation, a substitution lemma is required. Because variables are values and our language includes a value restriction (in the TBoxLV rule), we prove the substitution lemma only when the term being substituted in is a value.

Lemma 3.2.7 (Substitution)**.** If $\Gamma, x :^{\theta'} B \vdash^{\theta} a : A$ and $\Gamma \vdash^{\theta'} v : B$, then $\Gamma \vdash^{\theta} [v/x]a : A$.

The proof goes by induction on the first typing derivation and is entirely standard. Since we employ a call-by-value operational semantics, the value-restricted substitution lemma is enough for the beta-reduction cases of the preservation theorem.

Preservation also requires inversion lemmas which describe the types that can be given to certain term forms. Like the canonical forms lemmas in the previous section, the inversion lemmas are slightly complicated because the introduction of @-types is not marked in the syntax of terms. For example, we might expect the following lemma to hold:

Lemma. If $\Gamma \vdash^{\theta} \mathsf{inl}\, a : A$ then $A = B_1 + B_2$ for some types B_1 and B_2 such that $\Gamma \vdash^{\theta} a : B_1$.

However, this lemma is false because it may also the case that A has the form $(B_1 + B_2)@\theta_1$ or $((B_1 + B_2)@\theta_1)@\theta_2$, and so on. Additionally, the fragment in which a typechecks will depend on whether an @-type was used. The following modified inversion lemma can be proved by a straightforward induction on the typing derivation:

Lemma 3.2.8 (Inversion for inl)**.** Suppose $\Gamma \vdash^{\theta} \mathsf{inl}\, a : A$. Then either:

- $A = B_1 + B_2$ for some B_1 and B_2 such that $\Gamma \vdash^{\theta} a : B_1$,

- or $A = (...((B_1 + B_2)@\theta_1)...)@\theta_n$ for some B_1, B_2 and $\theta_1,...,\theta_n$ such that $\Gamma \vdash^{\theta_1} a : B_1$.

It is convenient to prove the following corollary for the common case that the type has the expected form. This arises in the proof of preservation.

Lemma 3.2.9 (Inversion for inl at sum types)**.** If $\Gamma \vdash^{\theta} \mathsf{inl}\, a : B_1 + B_2$ then $\Gamma \vdash^{\theta} a : B_1$.

After proving similar lemmas for the term forms $\mathsf{inr}\, a$, $\mathsf{rec} f\ x.b$ and $\mathsf{roll}\, a$, we are prepared for preservation itself.

Theorem 3.2.10 (Preservation)**.** If $\Gamma \vdash^{\theta} a : A$ and $a \rightsquigarrow a'$, then $\Gamma \vdash^{\theta} a' : A$.

Proof. By induction on the typing derivation, examining the possible forms of the reduction step in each case. □

3.3 Adapting the Girard–Tait Method

To motivate the use of step-indexed logical relations in our normalization proof, we will first revisit the standard Girard–Tait reducibility method [27, 58] and examine why more direct adaptations of it fail. Traditional techniques for proving strong normalization begin by defining the "interpretation" of each type. That is, for each type A, a set of terms $[\![A]\!]$ is defined approximating the type A and where each term in the set is known to be strongly normalizing. Then a theorem, called "soundness" or "the fundamental theorem of the interpretation", is proved. It shows that if a has type A then $a \in [\![A]\!]$. This implies a is strongly normalizing.

3.3.1 First Attempt: Ignoring the Programmatic Fragment

We begin by modifying this technique in two ways to fit our setting. First, since we have a deterministic call-by-value operational semantics, the interpretation of each type will be a set of values (not arbitrary terms). Second, since the terms at a given type differ in the programmatic and logical fragments, we index the interpretation by θ, writing $[\![A]\!]^{\theta}$.

Since we are uninterested in the normalization behavior of the programmatic fragment, it is natural to think that our model of it can be very simple. Perhaps, for example, just the values of the appropriate type will do. In this section, we will demonstrate that such an attempt fails. Consider the following interpretation:

$$\begin{array}{lcl}
[\![A]\!]^{\mathsf{P}} & = & \{v \mid \cdot \vdash^{\mathsf{P}} v : A\} \\
\\
[\![\mathsf{Unit}]\!]^{\mathsf{L}} & = & \{()\} \\
[\![A + B]\!]^{\mathsf{L}} & = & \{\mathsf{inl}\, v \mid v \in [\![A]\!]^{\mathsf{L}}\} \cup \{\mathsf{inr}\, v \mid v \in [\![B]\!]^{\mathsf{L}}\} \\
[\![A \to B]\!]^{\mathsf{L}} & = & \{\lambda x.a \mid \cdot \vdash^{\mathsf{L}} \lambda x.a : A \to B \\
 & & \qquad \text{and for any } v \in [\![A]\!]^{\mathsf{L}}, [v/x]a \leadsto^* v' \in [\![B]\!]^{\mathsf{L}}\} \\
[\![A@\theta]\!]^{\mathsf{L}} & = & \{v \mid v \in [\![A]\!]^{\theta}\} \\
[\![\mu\, \alpha.A]\!]^{\mathsf{L}} & = & \emptyset \\
[\![\alpha]\!]^{\mathsf{L}} & = & \emptyset
\end{array}$$

Here, the logical interpretation of Unit contains only $()$. The logical interpretation of a sum type $A+B$ contains $\mathsf{inl}\, v$ for every v in the interpretation A and $\mathsf{inr}\, v$ for every v in the interpretation of B. The logical interpretation of functions types is standard: $A \to B$ contains the term $\lambda x.a$ if, for any v in the interpretation of the domain, $[v/x]a$ reduces to a value in the interpretation of the range ("related functions take related arguments to related results"). The logical interpretation of $A@\theta$ comprises the values v where v is in $[\![A]\!]^{\theta}$. Finally, the logical interpretations of recursive types and type variables are empty, since these are used only in the programmatic fragment.

Before we can state a soundness theorem, we must account for contexts. We use ρ for mappings of variables to values, and write $\rho \models \Gamma$ if $x :^{\theta} A \in \Gamma$ implies $\rho x \in [\![A]\!]^{\theta}$. We let ρa stand for the simultaneous replacement of the variables in a by the corresponding terms in ρ.

In this setting, we would hope to be able to prove the following soundness theorem:

Soundness: Suppose $\Gamma \vdash^{\mathsf{L}} a : A$ and $\rho \models \Gamma$. Then $\rho a \leadsto^* v \in [\![A]\!]^{\mathsf{L}}$.

In a proof by induction on the typing derivation, most of the cases offer little resistance (the interested reader is encouraged to write out the case for the TLam and TApp rules). However, the proof gets stuck at the case for the mobile values rule:

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \mathsf{Mob}(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\ \textsc{TMobVal}$$

Here, we must show that $\rho v \in [\![A]\!]^{\mathsf{L}}$ (substituting values into a value produces a value, so ρv does not step). However, since the premise is in the programmatic fragment, we have no induction hypothesis for v. If $A = \mathsf{Unit}$, we can complete the case using a canonical forms lemma (since we know by a substitution lemma that $\cdot \vdash^{\mathsf{L}} \rho v : \mathsf{Unit}$). However, if A is $B@\mathsf{L}$, we are stuck.

3.3.2 Second Attempt: Partial Correctness for the Programmatic Fragment

Our previous attempt failed because the language permits values of mobile types to move from the programmatic fragment to the logical fragment, but the theorem we were trying to prove didn't capture any information about the programmatic fragment. To fix this, we might try making two changes. First, the programmatic and logical interpretations should agree at mobile types. Second, the programmatic interpretation and the soundness theorem should be modified to prove a partial correctness result for the programmatic fragment. In particular, since only values may be moved from the programmatic fragment to the logical fragment, we'll need to know that *if* a programmatic term normalizes, *then* it is in the appropriate interpretation.

These changes should allow us to handle the previously problematic TMobVal case. Consider the following modified interpretation, ignoring recursive types for the moment:

$$\begin{array}{lcl}
[\![\mathsf{Unit}]\!]^\theta & = & \{()\} \\
[\![A+B]\!]^\theta & = & \{\mathsf{inl}\,v \mid v \in [\![A]\!]^\theta\} \cup \{\mathsf{inr}\,v \mid v \in [\![B]\!]^\theta\} \\
[\![A \to B]\!]^{\mathsf{L}} & = & \{\lambda x.a \mid \cdot \vdash^{\mathsf{L}} \lambda x.a : A \to B \\
 & & \qquad \text{and for any } v \in [\![A]\!]^{\mathsf{L}}, [v/x]a \leadsto^* v' \in [\![B]\!]^{\mathsf{L}}\} \\
[\![A \to B]\!]^{\mathsf{P}} & = & \{\mathsf{rec}\,f\,x.a \mid \cdot \vdash^{\mathsf{P}} \mathsf{rec}\,f\,x.a : A \to B \\
 & & \qquad \text{and for any } v \in [\![A]\!]^{\mathsf{P}},\ \text{if } [v/x][\mathsf{rec}\,f\,x.a/f]a \leadsto^* v' \\
 & & \qquad\qquad\qquad\qquad \text{then } v' \in [\![B]\!]^{\mathsf{P}}\} \\
[\![A@\theta']\!]^\theta & = & \{v \mid v \in [\![A]\!]^{\theta'}\}
\end{array}$$

Here, the logical interpretation is unchanged. The programmatic interpretation of the mobile types is now the same as the logical interpretation. Finally, we have modified the programmatic interpretation of function types to state a partial correctness property: *if* a function terminates when passed a value in the interpretation of its domain, *then* the result must be in the interpretation of its range. We now restate the soundness theorem similarly.

Soundness: Suppose $\Gamma \vdash^\theta a : A$ and $\rho \models \Gamma$.

- If θ is L, then $\rho a \leadsto^* v \in [\![A]\!]^{\mathsf{L}}$.
- If θ is P and $\rho a \leadsto^* v$, then $v \in [\![A]\!]^{\mathsf{P}}$.

With the modified interpretation and soundness theorem, the TMobVal case now goes through. Because the rule only applies to values, the theorem now yields a useful induction hypothesis for the premise.

Unfortunately, this style of definition introduces a new problem: the programmatic interpretation of recursive types. The previous definition (from Section 3.3.1) is insufficient to handle the TUnroll case of the new soundness theorem. To extend our partial correctness property, we might demand that when unrolling results in a value, that value is in the interpretation of the unrolled type:

$$[\![\mu\,\alpha.A]\!]^{\mathsf{P}} \quad = \quad \{\mathsf{roll}\,v \mid \cdot \vdash^{\mathsf{P}} \mathsf{roll}\,v : \mu\,\alpha.A \text{ and } v \in [\![[\mu\,\alpha.A/\alpha]A]\!]^{\mathsf{P}}\}$$

However, this is not a valid definition. If the interpretation is a function defined by recursion on the structure of types, the substitution in $[\![[\mu\,\alpha.A/\alpha]A]\!]^{\mathsf{P}}$ ruins its well-foundedness.

A second, slightly harder to see problem also exists with this definition: a proof of the soundness theorem as stated above runs into trouble in the case for recursive functions. Consider the TREC case of a proof of that theorem by induction on typing derivations:

$$\frac{\begin{array}{l}\Gamma, y :^{\mathsf{P}} A, f :^{\mathsf{P}} A \to B \vdash^{\mathsf{P}} a : B \\ \mathsf{Mob}(A)\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\,f\ y.a : A \to B}\ \textsc{TRec}$$

Here, we must show that $\mathsf{rec}\,f\ y.\rho a \in [\![A \to B]\!]^{\mathsf{P}}$. But to do this we will need to use the induction hypothesis for the subderivation which checks a. Since the context for this subderivation is extended and $\mathsf{rec}\,f\ x.\rho a$ is substituted for f by the definition of the intepretation, the "$\rho \models \Gamma$" premise of the induction hypothesis would require us to show that $\mathsf{rec}\,f\ y.\rho a \in [\![A \to B]\!]^{\mathsf{P}}$. This is exactly what we were trying to show in the first place, so we are stuck. Thus, even if we removed recursive types from the system, the approach suggested here would fail.

3.3.3 A Step-Indexed Interpretation

Happily, a technique exists in the literature to cope with the circularity introduced by iso-recursive types and recursive functions. Step-indexed logical relations [2, 5] add an index to the interpretation, indicating the number of available future execution steps. Terms in the relation are guaranteed to respect the property in question only for the number of steps indicated. The interpretation is defined recursively on this additional index, circumventing the circularity problem we encountered above.

However, the usual formulation of a step-indexed type interpretation only lends itself to proving partial correctness properties—it tells us that an expression will not do anything bad for the next k steps. By contrast, in the normalization theorem we want to know that every expression will eventually do something good (namely reduce to a value). In our definition, we take a hybrid approach by only counting steps that happen in the P fragment. The difference can be seen by comparing the definitions of $\mathcal{V}[\![A \to B]\!]^{\mathsf{L}}_k$ and $\mathcal{V}[\![A \to B]\!]^{\mathsf{P}}_k$, which say "$j \leq k$" and "$j < k$" respectively. If all θs in a derivation are L, then no inequalities are strict, so the step-count k never needs to decrease.

Following Ahmed [2], our interpretation is split into two parts. The *value* interpretation $\mathcal{V}[\![A]\!]^{\theta}_k$ resembles the interpretations shown in the previous sections. The k

index here indicates that when a value appears in a larger term, its programmatic components will be "well behaved" for at least k steps of computation. The *computational* interpretation $\mathcal{C}[\![A]\!]^{\theta}_{k}$ contains closed terms, not just values. Its definition resembles the statement of the soundness theorem from the previous section, with steps counted explicitly. Terms in $\mathcal{C}[\![A]\!]^{\mathsf{L}}_{k}$ are guaranteed to normalize to values in $\mathcal{V}[\![A]\!]^{\mathsf{L}}_{k}$. On the other hand, we have a partial correctness property for terms in $\mathcal{C}[\![A]\!]^{\mathsf{P}}_{k}$—*if* they reach a value in j steps for some $j \le k$, *then* the value is in $\mathcal{V}[\![A]\!]^{\mathsf{P}}_{k-j}$.

$$
\begin{array}{lcl}
\mathcal{V}[\![\mathsf{Unit}]\!]^{\theta}_{k} & = & \{()\} \\
\mathcal{V}[\![A+B]\!]^{\theta}_{k} & = & \{\mathsf{inl}\, v \mid v \in \mathcal{V}[\![A]\!]^{\theta}_{k}\} \cup \{\mathsf{inr}\, v \mid v \in \mathcal{V}[\![B]\!]^{\theta}_{k}\} \\
\mathcal{V}[\![A@\theta']\!]^{\theta}_{k} & = & \{v \mid v \in \mathcal{V}[\![A]\!]^{\theta'}_{k}\} \\
\mathcal{V}[\![A \to B]\!]^{\mathsf{L}}_{k} & = & \{\lambda x.a \mid \cdot \vdash^{\mathsf{L}} \lambda x.a : A \to B \\
 & & \qquad \text{and } \forall j \le k,\ \text{if } v \in \mathcal{V}[\![A]\!]^{\mathsf{L}}_{j} \text{ then } [v/x]a \in \mathcal{C}[\![B]\!]^{\mathsf{L}}_{j}\} \\
\mathcal{V}[\![A \to B]\!]^{\mathsf{P}}_{k} & = & \{\mathsf{rec}\, f\ x.a \mid \cdot \vdash^{\mathsf{P}} \mathsf{rec}\, f\ x.a : A \to B \\
 & & \qquad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}[\![A]\!]^{\mathsf{P}}_{j} \text{ then } [v/x][\mathsf{rec}\, f\ x.a/f]a \in \mathcal{C}[\![B]\!]^{\mathsf{P}}_{j}\} \\
\mathcal{V}[\![\mu\,\alpha.A]\!]^{\mathsf{L}}_{k} & = & \emptyset \\
\mathcal{V}[\![\mu\,\alpha.A]\!]^{\mathsf{P}}_{k} & = & \{\mathsf{roll}\, v \mid \cdot \vdash^{\mathsf{P}} \mathsf{roll}\, v : \mu\,\alpha.A \text{ and } \forall j < k, v \in \mathcal{V}[\![[\mu\,\alpha.A/\alpha]A]\!]^{\mathsf{P}}_{j}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}[\![A]\!]^{\mathsf{P}}_{k} & = & \{a \mid \cdot \vdash^{\mathsf{P}} a : A \text{ and } \forall j \le k,\ \text{if } a \leadsto^{j} v \text{ then } v \in \mathcal{V}[\![A]\!]^{\mathsf{P}}_{(k-j)}\} \\
\mathcal{C}[\![A]\!]^{\mathsf{L}}_{k} & = & \{a \mid \cdot \vdash^{\mathsf{L}} a : A \text{ and } a \leadsto^{*} v \in \mathcal{V}[\![A]\!]^{\mathsf{L}}_{k}\}
\end{array}
$$

The value interpretation is similar to the proposed interpretation in the previous section, with two changes. First, the function type cases now refer to the computation interpretation rather than explicitly mentioning the reduction behavior. Second, the step indices track reductions in the programmatic fragment. In particular, note that the programmatic interpretation of function types demands that related functions take related arguments to related results at all *strictly smaller* indices, effectively counting the one beta reduction step that this definition unfolds. The beta step in the logical interpretation is not counted, since we are tracking only the reduction of programmatic components.

Unlike the proposed definition from the previous section, this interpretation is well defined. We can formalize its descending well-founded metric as a lexicographically ordered triple (k, A, Ω): k is the index, A is the type and Ω is one of $\mathcal{C}$ or $\mathcal{V}$ with $\mathcal{V} < \mathcal{C}$. The third element of the triple tracks which interpretation is being called—the computational interpretation may call the value interpretation at the same index and type, but not vice-versa.

3.3.4 Normalization

The step-indexed interpretation from the previous section repairs the problems encountered in the first two proposed interpretations and can be used to prove normalization for the logical fragment. Since our results are formalized in Coq, we give only a high-level overview of the proof here. To begin, we must update the $\rho \models \Gamma$ judgement to account for steps. We now write $\Gamma \models_k \rho$ when $x :^{\theta} A \in \Gamma$ implies $\rho x \in \mathcal{V}[\![A]\!]^{\theta}_{k}$.

Three key lemmas are needed in the main soundness theorem. The first is a standard "downward closure" property that often accompanies step-indexed logical relations. This lemma captures the idea that we build a more precise interpretation of a type by considering terms that must be valid for more steps.

Lemma 3.3.1 (Downward Closure)**.** For any A and θ, if $j \leq k$ then $\mathcal{V}[\![A]\!]^{\theta}_{k} \subseteq \mathcal{V}[\![A]\!]^{\theta}_{j}$ and $\mathcal{C}[\![A]\!]^{\theta}_{k} \subseteq \mathcal{C}[\![A]\!]^{\theta}_{j}$.

Two lemmas relate the programmatic and logical interpretations, corresponding to the TMobVal and TSub typing rules. The first says that the two interpretations agree on mobile types:

Lemma 3.3.2. If $\mathsf{Mob}(A)$, then $\mathcal{V}[\![A]\!]^{\mathsf{L}}_{k} = \mathcal{V}[\![A]\!]^{\mathsf{P}}_{k}$.

The second captures the idea that the logical fragment is a subsystem of the programmatic fragment:

Lemma 3.3.3. For any A and k, $\mathcal{V}[\![A]\!]^{\mathsf{L}}_{k} \subseteq \mathcal{V}[\![A]\!]^{\mathsf{P}}_{k}$ and $\mathcal{C}[\![A]\!]^{\mathsf{L}}_{k} \subseteq \mathcal{C}[\![A]\!]^{\mathsf{P}}_{k}$.

The content of the soundness theorem is essentially the same as in our second failed attempt, but we can now state it more directly, using the computational interpretation. The theorem is proved by induction on step count k (to handle the case of recursive functions), with a nested induction on the typing derivation. It uses the lemmas outlined above.

Theorem 3.3.4 (Soundness)**.** If $\Gamma \vdash^{\theta} a : A$ and $\Gamma \models_k \rho$, then $\rho a \in \mathcal{C}[\![A]\!]^{\theta}_{k}$.

The normalization of the logical fragment is a direct consequence of this theorem and the definition of the interpretation.

Lemma 3.3.5 (Normalization)**.** If $\cdot \vdash^{\mathsf{L}} a : A$ then there exists a value v such that $a \leadsto^{*} v$.

Chapter 4

Adding Dependent Types

Head down towards Kansas.
We will get there when we get there, don't you worry.
Feel bad about the things we do along the way,
But not really that bad.

Psalms 40:2
The Mountain Goats

In this chapter, we will extend the language λ^θ with dependent types and equality. We call this new system LF$^\theta$, to reflect that it has roughly the same level of expressiveness as LF [31]. Like LF, this system has dependent types but not type-level computation or polymorphism. However, there are some important differences.

In being more precise about how LF$^\theta$ relates to LF, it is convenient to refer to Barendregt's lambda cube [6]. In particular, LF comprises the $(\star, \star)$ and $(\star, \Box)$ corners of the lambda cube. The former allows normal term-level functions, while the latter allows type-level functions that take terms as arguments ("dependent types"). It does not include the $(\Box, \star)$ corner (which allows polymorphic functions) or the $(\Box, \Box)$ corner (which allows functions from types to types, and is often referred to as type-level computation). On the other hand, LF$^\theta$ does not allow arbitrary functions from terms to types via the $(\star, \Box)$ rule, but we still say it has dependent types because we include an explicit type constructor that takes terms as arguments (equality) and because the range of an arrow type may refer to the value of its domain (i.e., arrow types have the form $(x : A) \to B$ rather than $A \to B$).

Unlike LF, LF$^\theta$ includes the logical and programmatic fragments we have seen previously, along with a novel equality type. In particular, the elimination of equality is not marked in the syntax of terms, a substantial departure from traditional intensional dependently typed languages where equality typically arises as a datatype with a pattern-matching elimination form. We saw in Chapter 2 that this unmarked elimination can be very convenient for programmers, as uses of conversion no longer "clutter" terms. However, as we will see in this chapter and the next two, this notion

Expressions

$$
\begin{array}{lll}
a,\ b,\ A,\ B & ::= & \star \mid (x\!:\!A) \to B \mid a = b \mid \mathsf{Nat} \mid A + B \mid \Sigma x\!:\!A.B \mid \mu x.A \mid A@\theta \\
& & \mid x \mid \lambda x.b \mid \mathsf{rec}\, f\, x.b \mid \mathsf{ind}\, f\, x.b \mid b\ a \mid \mathsf{refl} \mid \mathsf{inl}\, a \mid \mathsf{inr}\, b \\
& & \mid \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ y \Rightarrow a_2\} \mid \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} \\
& & \mid \langle a, b\rangle \mid \mathsf{Z} \mid \mathsf{S}\ a \mid \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \\
& & \mid \mathsf{roll}\, a \mid \mathsf{unroll}\, a
\end{array}
$$

Consistency Classifiers

$$
\theta \quad ::= \quad \mathsf{L} \mid \mathsf{P}
$$

Values

$$
\begin{array}{lll}
v & ::= & \star \mid (x\!:\!A) \to B \mid a = b \mid \mathsf{Nat} \mid A + B \mid \Sigma x\!:\!A.B \mid \mu x.A \mid A@\theta \\
& & \mid x \mid \lambda x.a \mid \mathsf{rec}\, f\, x.a \mid \mathsf{ind}\, f\, x.a \mid \mathsf{refl} \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \langle v_1, v_2\rangle \mid \mathsf{Z} \mid \mathsf{S}\ v \mid \mathsf{roll}\, v
\end{array}
$$

Figure 4.1: LF$^\theta$: Syntax

of equality substantially complicates the metatheory of the language.

We begin in Section 4.1 with a specification of LF$^\theta$. The type safety of LF$^\theta$ must be approached slightly differently than that of λ^θ. In particular, in Section 4.2 we will prove preservation and show that a proof of progress must wait until after we have demonstrated LF$^\theta$'s consistency. While LF$^\theta$'s notion of equality complicates the proof of type safety, we will see in Section 4.3 that the partially step-indexed technique from Chapter 3 can be extended in a standard way to show that the logical fragment of LF$^\theta$ normalizes (and is therefore consistent). Finally, in Section 4.4, we show progress for LF$^\theta$, completing the proof of type safety.

As was the case in Chapter 3, LF$^\theta$ and its metatheory have been completely mechanized in Coq. The proofs are available in this thesis's digital appendix [13]. For this reason, we focus here on a clear explanation of our techniques and the key differences between LF$^\theta$ and more traditional language, rather than explicitly writing out the details of every lemma.

4.1 The LF$^\theta$ Language

We begin our technical development with a specification of LF$^\theta$. This language extends λ^θ from Chapter 3 with dependent types in the form of an equality type and dependent pairs. We also include natural numbers and an associated terminating recursor. The syntax of LF$^\theta$ is shown in Figure 4.1. For uniformity, terms, types and the single kind $\star$ (the "type" of types) are drawn from the same syntactic category,

Evaluation Contexts

$$\begin{array}{lll} E & ::= & [\cdot] \mid E\, b \mid v\, E \mid \mathsf{inl}\, E \mid \mathsf{inr}\, E \mid \langle E, b\rangle \mid \langle v, E\rangle \mid \mathsf{S}\, E \mid \mathsf{roll}\, E \mid \mathsf{unroll}\, E \\ & & \mid \mathsf{scase}_z\ E\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \mid \mathsf{pcase}_z\ E\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} \\ & & \mid \mathsf{ncase}_E\ E\ \mathsf{of}\ \{Z \Rightarrow a; \mathsf{S}\ x \Rightarrow b\} \end{array}$$

$$\boxed{a \rightsquigarrow b}$$

$$\frac{}{(\mathsf{rec}\, f\, x.a)\ v \rightsquigarrow [v/x][\mathsf{rec}\, f\, x.a/f]a}\ \textsc{SFun} \qquad \frac{}{(\mathsf{ind}\, f\, x.a)\ v \rightsquigarrow [v/x][\lambda y.\lambda z.(\mathsf{ind}\, f\, x.a)\ y/f]a}\ \textsc{SInd}$$

$$\frac{}{(\lambda x.a)\ v \rightsquigarrow [v/x]a}\ \textsc{SLam} \qquad \frac{}{\mathsf{pcase}_z\ \langle v_1, v_2\rangle\ \mathsf{of}\ \{(x,\ y) \Rightarrow a\} \rightsquigarrow [\mathsf{refl}/z][v_1/x][v_2/y]a}\ \textsc{SCP}$$

$$\frac{}{\mathsf{scase}_z\ \mathsf{inl}\, v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_1}\ \textsc{SCL}$$

$$\frac{}{\mathsf{scase}_z\ \mathsf{inr}\, v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_2}\ \textsc{SCR}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{Z}\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z]a_1}\ \textsc{SCZ}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{S}\ v\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_2}\ \textsc{SCS}$$

$$\frac{}{\mathsf{unroll}\,(\mathsf{roll}\, v) \rightsquigarrow v}\ \textsc{SUnroll} \qquad \frac{a \rightsquigarrow b}{E[a] \rightsquigarrow E[b]}\ \textsc{SCtx}$$

Figure 4.2: LF$^\theta$: Operational Semantics

as in pure type systems [6].[1] By convention, we use lowercase metavariables a, b for expressions that are terms and uppercase metavariables A, B for expressions that are types.

The first line of the expression grammar lists the types, which now include the equality form $a = b$, dependent pairs $\Sigma x : A.B$, and Nat. The language's terms include those of λ^θ with the addition of natural numbers (constructed by Z and $\mathsf{S}\ a$ and eliminated by ncase), dependently typed pairs (constructed by $\langle a, b\rangle$ and eliminated by pcase), functions defined by natural number induction (constructed by $\mathsf{ind}\ f\ x.a$) and a primitive proof of equality (refl). The z subscript on the pattern-matching elimination forms will be explained along with the corresponding typing rules. Values include the term introduction forms as well as $\star$, all type forms, and variables. Including variables is safe, as it was in λ^θ, because CBV evaluation only

[1]Unfortunately, we will be forced to abandon a collapsed syntax in the next chapter for metatheoretic reasons (as described in detail in Section 6.4). This chapter retains a collapsed syntax in order to demonstrate that the problems we encounter later do not occur without the addition of polymorphism and type-level computation.

substitutes values for variables.

It is worth emphasizing that functions in LF$^\theta$ can be given dependent types $(x : A) \to B$. In such types, the result type B can depend on the value x of the argument. This value can appear inside equality types $a = b$, which are assertions that the term a equals the term b. The equality type has a trivial proof refl, which holds when two expressions can be reduced to the same term. Equality proofs can be eliminated implicitly, substituting provably equal terms at any point in a typing derivation.

Figure 4.2 gives a call-by-value, small-step operational semantics for LF$^\theta$ via the reduction relation $a \rightsquigarrow b$. The slightly unusual beta rule for natural number induction (SIND) is similar to the induction rule from THETA and is described in Section 4.1.1.

In the remainder of this section, we describe the typing judgement for LF$^\theta$. We begin with rules for the basic components of functional programming and rules for the interactions between the two fragments (Section 4.1.1). In both cases the rules closely resemble those from λ^θ, extended with dependent types and extra term forms. We conclude with a description of LF$^\theta$'s equality (Section 4.1.2).

4.1.1 Typing Basics

Like λ^θ, LF$^\theta$'s typing judgement $\Gamma \vdash^\theta a : A$ is indexed by a consistency classifier θ. The judgement is designed so that expressions that typecheck at L always terminate, but this is not the case for expressions that typecheck at P.

Figure 4.3 shows the typing rules for the basic building blocks of the language—variables, functions and various data structures and their types. Because we work with a collapsed syntax, we use the type system to identify which expressions are types: A is a well-formed type if $\Gamma \vdash^\theta A : \star$. Unlike THETA, LF$^\theta$ does not demand that all types check in the logical fragment. Instead, we require that if a term is checked in a given fragment, its type checks in the same fragment. As described in Section 6.1, this relaxed regime would introduce substantial problems for the metatheory of a larger system, but as we will see it does not complicate LF$^\theta$'s metatheory.

Contexts are lists of assumptions about the types of variables.

$$\Gamma ::= \emptyset \mid \Gamma, x :^\theta A$$

Each variable in the context is tagged with θ to indicate its fragment, and this tag is checked in the TVAR typing rule. A context is *valid*, written $\vdash \Gamma$, if each type A is valid in the corresponding fragment.

The rules TARR, TSIGMA, TSUM, and TMU check types for well-kindedness. For example, TARR checks a function type by checking the the domain and range. The premise Mob(A) has the same meaning here as in λ^θ. It is defined in Figure 4.4, which is discussed below.

There are three ways to define functions in LF$^\theta$. Rule TLAM types non-recursive λ-expressions in the logical fragment, whereas rule TREC types general recursive rec-expressions and can only be used in the programmatic fragment.

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

$$\dfrac{}{\vdash \cdot}\ \textsc{CNil} \qquad \dfrac{\vdash \Gamma}{\vdash \Gamma, x :^{\theta} \star}\ \textsc{CStar} \qquad \dfrac{\vdash \Gamma \quad \Gamma \vdash^{\theta} A : \star}{\vdash \Gamma, x :^{\theta} A}\ \textsc{CType}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\dfrac{(x :^{\theta} A) \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash^{\theta} x : A}\ \textsc{TVar} \qquad \dfrac{\begin{array}{c}\Gamma \vdash^{\theta} A : \star \quad \mathsf{Mob}\,(A)\\ \Gamma, x :^{\theta} A \vdash^{\theta} B : \star\end{array}}{\Gamma \vdash^{\theta} (x\!:\!A) \to B : \star}\ \textsc{TArr}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash^{\theta} b : (x\!:\!A) \to B\\ \Gamma \vdash^{\theta} a : A \quad \Gamma \vdash^{\theta} [a/x]B : \star\end{array}}{\Gamma \vdash^{\theta} b\ a : [a/x]B}\ \textsc{TApp} \qquad \dfrac{\begin{array}{c}\Gamma, f :^{\mathsf{P}} (x\!:\!A) \to B, x :^{\mathsf{P}} A \vdash^{\mathsf{P}} b : B\\ \Gamma \vdash^{\mathsf{P}} (x\!:\!A) \to B : \star\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\, x.b : (x\!:\!A) \to B}\ \textsc{TRec}$$

$$\dfrac{\begin{array}{c}\Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} b : B\\ \Gamma \vdash^{\mathsf{L}} (x\!:\!A) \to B : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} \lambda x.b : (x\!:\!A) \to B}\ \textsc{TLam} \qquad \dfrac{\begin{array}{l}\Gamma, x :^{\mathsf{L}} \mathsf{Nat}, f :^{\mathsf{L}} (y\!:\!\mathsf{Nat}) \to (z\!:\!\mathsf{S}\ y = x) \to B \vdash^{\mathsf{L}} b : B\\ \Gamma \vdash^{\mathsf{L}} (x\!:\!\mathsf{Nat}) \to B : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ind}\, f\, x.b : (x\!:\!\mathsf{Nat}) \to B}\ \textsc{TInd}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash^{\theta} a : A\\ \Gamma \vdash^{\theta} A + B : \star\end{array}}{\Gamma \vdash^{\theta} \mathsf{inl}\, a : A + B}\ \textsc{TInl} \qquad \dfrac{\begin{array}{c}\Gamma \vdash^{\theta} b : B\\ \Gamma \vdash^{\theta} A + B : \star\end{array}}{\Gamma \vdash^{\theta} \mathsf{inr}\, b : A + B}\ \textsc{TInr} \qquad \dfrac{\begin{array}{c}\Gamma \vdash^{\theta} A : \star\\ \Gamma \vdash^{\theta} B : \star\end{array}}{\Gamma \vdash^{\theta} A + B : \star}\ \textsc{TSum}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash^{\theta} a : (A_1 + A_2)@\theta' \quad \Gamma \vdash^{\theta} B : \star\\ \Gamma, x :^{\theta'} A_1, z :^{\mathsf{L}} \mathsf{inl}\, x = a \vdash^{\theta} b_1 : B\\ \Gamma, x :^{\theta'} A_2, z :^{\mathsf{L}} \mathsf{inr}\, x = a \vdash^{\theta} b_2 : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1; \mathsf{inr}\ x \Rightarrow b_2\} : B}\ \textsc{TSCase} \qquad \dfrac{\begin{array}{l}\Gamma \vdash^{\theta} \Sigma x\!:\!A.B : \star \quad \Gamma \vdash^{\theta} a : A\\ \Gamma \vdash^{\theta} b : [a/x]B\\ \Gamma \vdash^{\theta} [a/x]B : \star\end{array}}{\Gamma \vdash^{\theta} \langle a, b \rangle : \Sigma x\!:\!A.B}\ \textsc{TPair}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash^{\theta} A : \star \quad \mathsf{Mob}\,(A)\\ \Gamma, x :^{\theta} A \vdash^{\theta} B : \star\end{array}}{\Gamma \vdash^{\theta} \Sigma x\!:\!A.B : \star}\ \textsc{TSigma} \qquad \dfrac{\begin{array}{c}\Gamma \vdash^{\theta} a : (\Sigma x\!:\!A_1.A_2)@\theta' \quad \Gamma \vdash^{\theta} B : \star\\ \Gamma, x :^{\theta'} A_1, y :^{\theta'} A_2, z :^{\mathsf{L}} \langle x, y \rangle = a \vdash^{\theta} b : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} : B}\ \textsc{TPCase}$$

$$\dfrac{\Gamma, x :^{\mathsf{L}} \star \vdash^{\mathsf{L}} A : \star}{\Gamma \vdash^{\mathsf{L}} \mu x.A : \star}\ \textsc{TMu} \qquad \dfrac{\begin{array}{c}\Gamma \vdash^{\theta} a : [\mu x.A/x]A\\ \Gamma \vdash^{\theta} \mu x.A : \star\end{array}}{\Gamma \vdash^{\theta} \mathsf{roll}\, a : \mu x.A}\ \textsc{TRoll} \qquad \dfrac{\begin{array}{c}\Gamma \vdash^{\mathsf{P}} a : \mu x.A\\ \Gamma \vdash^{\mathsf{P}} [\mu x.A/x]A : \star\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{unroll}\, a : [\mu x.A/x]A}\ \textsc{TUnroll}$$

$$\dfrac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Nat} : \star}\ \textsc{TNat} \qquad \dfrac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Z} : \mathsf{Nat}}\ \textsc{TZero} \qquad \dfrac{\Gamma \vdash^{\theta} a : \mathsf{Nat}}{\Gamma \vdash^{\theta} \mathsf{S}\ a : \mathsf{Nat}}\ \textsc{TSucc}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash^{\theta} a : \mathsf{Nat} \quad \Gamma \vdash^{\theta} B : \star\\ \Gamma, z :^{\mathsf{L}} \mathsf{Z} = a \vdash^{\theta} b_1 : B\\ \Gamma, x :^{\theta} \mathsf{Nat}, z :^{\mathsf{L}} (\mathsf{S}\ x) = a \vdash^{\theta} b_2 : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow b_1; \mathsf{S}\ x \Rightarrow b_2\} : B}\ \textsc{TNCase}$$

Figure 4.3: LF$^{\theta}$ Typing: Variables, Functions, and Datatypes

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\ \textsc{TSub} \qquad \frac{\Gamma \vdash^{\theta'} A : \star}{\Gamma \vdash^{\theta} A@\theta' : \star}\ \textsc{TAt} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\theta} v : A@\theta' \\ \Gamma \vdash^{\theta'} A : \star\end{array}}{\Gamma \vdash^{\theta'} v : A}\ \textsc{TUnboxVal}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\theta} a : A \\ \Gamma \vdash^{\theta} A : \star\end{array}}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}\ \textsc{TBoxP} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathsf{L}} a : A \\ \Gamma \vdash^{\theta} A : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} a : A@\theta}\ \textsc{TBoxL} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathsf{P}} v : A \\ \Gamma \vdash^{\mathsf{P}} A : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\ \textsc{TBoxLV}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \Gamma \vdash^{\mathsf{L}} A : \star \quad \mathsf{Mob}\,(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\ \textsc{TMVal}$$

$$\boxed{\mathsf{Mob}\,(A)}$$

$$\frac{}{\mathsf{Mob}\,(A@\theta)}\ \textsc{MAt} \qquad \frac{}{\mathsf{Mob}\,(a = b)}\ \textsc{MEq} \qquad \frac{\mathsf{Mob}\,(A) \quad \mathsf{Mob}\,(B)}{\mathsf{Mob}\,(\Sigma x{:}A.B)}\ \textsc{MSigma}$$

$$\frac{}{\mathsf{Mob}\,(\mathsf{Nat})}\ \textsc{MNat} \qquad \frac{\mathsf{Mob}\,(A) \quad \mathsf{Mob}\,(B)}{\mathsf{Mob}\,(A + B)}\ \textsc{MSum}$$

Figure 4.4: LF$^{\theta}$ Typing: The Fragments

Additionally, terminating recursion over natural numbers is provided in the logical fragment by rule TInd. When typechecking the body of a terminating recursive function ($\mathsf{ind}\ f\ x.b$), the recursive call f takes an extra argument proving that it is being applied to the predecessor of the initial argument x. This ensures termination. When beta-reducing such an expression, this argument is ignored by wrapping the function in an extra lambda (rule SInd from Figure 4.2).

The rule for function application, TApp, differs from the usual application rule in pure dependently typed languages in the additional third premise $\Gamma \vdash^{\theta} [a/x]B : \star$, which checks that the result type is well-formed. This mirrors the application rule in Theta, described in Section 2.2.1.

The rules for sum types (TSum, TInl, TInr, and TSCase) provide dependent case analysis. The term scase binds the logical variable z inside both branches of the case. As we saw in Chapter 2, this variable provides an equality between the scrutinee and the pattern of the branch so that typechecking is flow-sensitive. At runtime, this variable is replaced by refl because the scrutinee matches the pattern for the branch to be taken. Additionally, as in λ^{θ}, we use the @ type constructor to allow each fragment to pattern match on expressions from the other. The typing rules for @ in LF$^{\theta}$ appear in Figure 4.4, and are described below.

The rules for dependent products (TSigma, TPair, TPCase) allow the type of the second component of the pair to depend on the value of the first component. The first component is required to have a mobile type, for reasons similar to function domains (this is somewhat less restrictive than THETA, where we required all datatype arguments to be mobile). As with function application, the premise $\Gamma \vdash^{\theta} [a/x]B : \star$ ensures that substituting the expression a does not violate any assumptions made about the value x in the type of the second component. Analogously to sums, the eliminator for pairs makes available a logical proof z that equates the scrutinee to the pattern in the body of the match. The availability of this equality means that the strong elimination forms (projections) for Σ-types are derivable.

The rules TMU, TROLL and TUNROLL deal with general recursive types. These are the standard rules for iso-recursive types (see, e.g., [50]). But recursive types with negative occurrences—that is, with the recursive variable appearing to the left of an arrow, such as $\mu x.(x \to \mathsf{Nat})$—are a potential source of nontermination. To ensure normalization, it suffices to restrict the the elimination rule TUNROLL to be in P. The introduction rule TROLL can be used in both fragments. This reflects the fact that it is not dangerous to *construct* negative datatype values; the potential nontermination comes from their elimination.

The typing rules relating the two fragments appear in Figure 4.4. These closely mirror the rules we saw in LF$^{\theta}$ and THETA. Rule TSUB formalizes the idea that the logical fragment is a sublanguage of the programmatic fragment. Rule TAT checks @-types, while rules TBOXP, TBOXL and TBOXLV introduce them and rule TUNBOXVAL eliminates them. Rule TMVAL allows mobile values to be used in the logical fragment even if they were computed programmatically. The restrictions in these rules are identical to the ones we saw in previous systems and have the same motivations.

4.1.2 Reasoning About Equality

The rules for LF$^{\theta}$'s propositional equality (Figure 4.5) are similar to those we saw in the context of THETA. As in that system, a major design goal is to permit writing proofs *about* potentially non-terminating programs. Thus, the rule TEQ shows that the type $a = b$ is well-formed and in the logical fragment even when a and b can be typechecked only programmatically.

As in THETA, the primitive proof of equality refl is checked using *parallel reduction* $a \Rrightarrow b$ rather than our previously defined call-by-value relation $a \rightsquigarrow b$. Parallel reduction has the advantage of equating more terms, and also simplifies the proof of preservation. The relation $\Rrightarrow$ can only reduce redexes when the active subexpression is a value, so the rule TREFL proves $(\lambda x.a)\ v = [v/x]a$ but not the more general $(\lambda x.a)\ b = [b/x]a$. This value restriction reflects the usual equational theory of a CBV language. The complete definition of $a \Rrightarrow b$ can be found in Appendix A.

Equality proofs are used to modify types by rule TCONV. The equality proof is checked in L to ensure that it is valid, since all types are inhabited in the programmatic fragment. As in TAPP, we need to check that b_2 does not violate any value restrictions,

Head Forms

$hf \quad ::= \quad \mathsf{HStar} \mid \mathsf{HArr} \mid \mathsf{HAt}\,\theta \mid \mathsf{HEq} \mid \mathsf{HNat} \mid \mathsf{HSum} \mid \mathsf{HSigma} \mid \mathsf{HMu}$

$$\boxed{\mathsf{hd}\,(A) = hf}$$

$$\frac{}{\mathsf{hd}\,(\star) = \mathsf{HStar}}\ \textsc{HStar} \qquad \frac{}{\mathsf{hd}\,((x\!:\!A) \to B) = \mathsf{HArr}}\ \textsc{HArr} \qquad \frac{}{\mathsf{hd}\,(A@\theta) = \mathsf{HAt}\,\theta}\ \textsc{HAt}$$

$$\frac{}{\mathsf{hd}\,(A = B) = \mathsf{HEq}}\ \textsc{HEq} \qquad \frac{}{\mathsf{hd}\,(\mathsf{Nat}) = \mathsf{HNat}}\ \textsc{HNat} \qquad \frac{}{\mathsf{hd}\,(A + B) = \mathsf{HSum}}\ \textsc{HSum}$$

$$\frac{}{\mathsf{hd}\,(\Sigma x\!:\!A.B) = \mathsf{HSigma}}\ \textsc{HSigma} \qquad \frac{}{\mathsf{hd}\,(\mu x.A) = \mathsf{HMu}}\ \textsc{HMu}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \quad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a = b : \star}\ \textsc{TEq} \qquad \frac{\begin{array}{c} a \Rrightarrow^{*} c \quad b \Rrightarrow^{*} c \\ \Gamma \vdash^{\theta_1} a : A \quad \Gamma \vdash^{\theta_2} b : B \end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = b}\ \textsc{TRefl}$$

$$\frac{\begin{array}{l} \Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \quad \Gamma \vdash^{\theta} a : [b_1/x]A \\ \Gamma \vdash^{\theta} [b_2/x]A : \star \end{array}}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \textsc{TConv} \qquad \frac{\begin{array}{l} \Gamma \vdash^{\mathsf{L}} a_1 : B_1 = B_2 \\ \mathsf{hd}(B_1) \neq \mathsf{hd}(B_2) \\ \Gamma \vdash^{\theta} a : A \\ \Gamma \vdash^{\theta} A : \star \quad \Gamma \vdash^{\theta'} B : \star \end{array}}{\Gamma \vdash^{\theta'} a : B}\ \textsc{TContra}$$

Figure 4.5: LF$^{\theta}$ Typing: Equality

so the last premise checks the well-formedness of the type given to the converted term. Uses of equality are not marked in the term and thus do not interfere with reduction.

Rule TCONTRA eliminates contradictory equalities, and is somewhat more general than its THETA counterpart. In particular, TCONTRA in LF$^{\theta}$ can be used to give any term any type in either fragment, as long as the term has some type and the new type is well-formed in the relevant fragment. THETA's version has almost the same expressive power, since it can be used to check equality proofs that can then change the type of arbitrary terms. It cannot, however, change the fragment in which a given term checks. This additional bit of expressiveness is convenient in the preservation proof.

We also include several injectivity axioms for the type constructors of LF$^{\theta}$, pictured in Figure 4.6. These axioms are used in the proof of the inversion lemmas needed by preservation, as we will see in Section 4.2.2. We do not include new term forms to make use of these rules. Instead they are unmarked, much like rule TCONV.

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : ((x\!:\!A_1) \to A_2) = ((x\!:\!B_1) \to B_2) \\ \Gamma \vdash^{\theta} A_1 = B_1 : \star\end{array}}{\Gamma \vdash^{\theta} a : A_1 = B_1}\ \textsc{TArrInv1}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : ((x\!:\!A_1) \to A_2) = ((x\!:\!B_1) \to B_2) \\ \Gamma \vdash^{\theta'} v : A_1 \quad \Gamma \vdash^{\theta} [v/x]A_2 = [v/x]B_2 : \star\end{array}}{\Gamma \vdash^{\theta} a : [v/x]A_2 = [v/x]B_2}\ \textsc{TArrInv2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (A_1 + A_2) = (B_1 + B_2) \\ \Gamma \vdash^{\theta} A_1 = B_1 : \star\end{array}}{\Gamma \vdash^{\theta} a : A_1 = B_1}\ \textsc{TSumInv1} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (A_1 + A_2) = (B_1 + B_2) \\ \Gamma \vdash^{\theta} A_2 = B_2 : \star\end{array}}{\Gamma \vdash^{\theta} a : A_2 = B_2}\ \textsc{TSumInv2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\Sigma x\!:\!A_1.A_2) = (\Sigma x\!:\!B_1.B_2) \\ \Gamma \vdash^{\theta} A_1 = B_1 : \star\end{array}}{\Gamma \vdash^{\theta} a : A_1 = B_1}\ \textsc{TSigmaInv1}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\Sigma x\!:\!A_1.A_2) = (\Sigma x\!:\!B_1.B_2) \\ \Gamma \vdash^{\theta'} v : A_1 \quad \Gamma \vdash^{\theta} [v/x]A_2 = [v/x]B_2 : \star\end{array}}{\Gamma \vdash^{\theta} a : [v/x]A_2 = [v/x]B_2}\ \textsc{TSigmaInv2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\mu x.A) = (\mu x.B) \\ \Gamma \vdash^{\theta} ([\mu x.A/x]A) = ([\mu x.B/x]B) : \star\end{array}}{\Gamma \vdash^{\theta} a : ([\mu x.A/x]A) = ([\mu x.B/x]B)}\ \textsc{TMuInv} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (A@\theta') = (B@\theta') \\ \Gamma \vdash^{\theta} A = B : \star\end{array}}{\Gamma \vdash^{\theta} a : A = B}\ \textsc{TAtInv}$$

Figure 4.6: LF$^{\theta}$ Typing: Injectivity Axioms

4.2 Preservation and a Problem for Progress

We now begin analyzing the metatheory of LF$^{\theta}$. We are interested in two properties. First, that the entire language is type safe, including both the L and P fragments. Second, that any closed term in the L fragment normalizes, which implies logical consistency.

Before beginning the proof of type safety, it is convenient to observe a few basic facts about typing in LF$^{\theta}$. First, the kind $\star$ itself has no type. This fact is useful to eliminate a number of contradictory cases in the proofs that follow. It may be proved by an induction on typing derivations.

Lemma 4.2.1 ($\star$ has no type)**.** There is no derivation of $\Gamma \vdash^{\theta} \star : A$ for any Γ, θ or A.

Secondly, LF$^{\theta}$ enjoys a standard "regularity" property which helps keep track of the various "levels" of expressions.

Lemma 4.2.2 (Regularity)**.** If $\Gamma \vdash^{\theta} a : A$, then either $A = \star$ or $\Gamma \vdash^{\theta} A : \star$.

The regularity result depends on a number of basic lemmas, including weakening and several simple results about the interactions between the fragments. These details are simple to work out and we elide them. They are provided in detail for the system PCC$^{\theta}$ in Chapter 5, and the same approach used there works here.

4.2.1 Getting Stuck on Progress

Type safety will be proved using progress and preservation lemmas. For λ^{θ}, we began with progress. However, for LF$^{\theta}$, this is no longer possible. In particular, the canonical forms lemmas can not be proved until after we have shown LF$^{\theta}$ to be consistent. Consider a standard canonical forms lemma for sums:

Lemma (Canonical forms for sums)**.** If $\cdot \vdash^{\theta} v : B_1 + B_2$ then there is some v' such that $v = \mathsf{inl}\, v'$ or $v = \mathsf{inr}\, v'$.

This lemma is usually proved by induction on the typing derivation. As described in Section 3.2.1, the fact that introductions and eliminations of @-types are not marked in the syntax requires us to generalize this lemma as follows:

Lemma (Generalized canonical forms for sums)**.** Suppose $\cdot \vdash^{\theta} v : A$ and $A = B_1 + B_2$ or $A = (...((B_1 + B_2)@\theta_1)...)@\theta_n$ for some $\theta_1, ..., \theta_n$. Then there is some v' such that $v = \mathsf{inl}\, v'$ or $v = \mathsf{inr}\, v'$.

However, in the case of LF$^{\theta}$, even proving this generalized form by induction on the typing derivation will get stuck. The problem occurs if the derivation goes by rule TConv:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \quad \Gamma \vdash^{\theta} a : [b_1/x]A \\ \Gamma \vdash^{\theta} [b_2/x]A : \star\end{array}}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \textsc{TConv}$$

Here, it may be the case that $[b_2/x]A$ has the form $B_1 + B_2$ while $[b_1/x]A$ does not. For example, if there is a proof of $\mathsf{Nat} = (\mathsf{Nat} + \mathsf{Nat})$ in the system, then TConv can be used to prove that $\cdot \vdash^{\mathsf{L}} \mathsf{Z} : \mathsf{Nat} + \mathsf{Nat}$. So, the canonical forms lemma will not hold unless the system is consistent. Accordingly, we delay the proof of progress until after we have demonstrated that the logical fragment of LF$^{\theta}$ normalizes and is consistent.

4.2.2 Substitution and Inversion

As usual, the preservation proof relies on substitution and inversion lemmas. As we saw in Chapter 3 with λ^{θ}, the substitution lemma is restricted to values due to the value restrictions in the type system:

Lemma 4.2.3 (Substitution)**.** If $\Gamma_1, x :^{\theta'} B, \Gamma_2 \vdash^{\theta} a : A$ and $\Gamma_1 \vdash^{\theta'} v : B$, then $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta} [v/x]a : [v/x]A$.

Preservation requires inversion lemmas for most term forms. Due to our collapsed syntax and the fact that our typing rules have been designed to maintain regularity, inversion lemmas for term forms often require corresponding inversion lemmas for type forms. The inversion lemmas for types have the expected statements, and their proofs are relatively straightforward. We show two examples:

Lemma 4.2.4 (Inversion for sum types)**.** If $\Gamma \vdash^{\theta} A_1 + A_2 : B$ then $\Gamma \vdash^{\mathsf{L}} A_1 : \star$, $\Gamma \vdash^{\mathsf{L}} A_2 : \star$, and $B = \star$.

Lemma 4.2.5 (Inversion for pair types)**.** If $\Gamma \vdash^{\theta} \Sigma x : A_1.A_2 : B$ then $\mathsf{Mob}\,(A_1)$, $B = \star$, $\Gamma \vdash^{\theta} A_1 : \star$, and $\Gamma, x :^{\theta} A_1 \vdash^{\theta} A_2 : \star$.

As in Chapter 3, the inversion lemmas for terms in LF$^{\theta}$ will be somewhat more complicated than usual to cope with the unmarked introduction and eliminations of @-types. There is an additional twist related to the unmarked eliminations of equalities that is new to LF$^{\theta}$. Consider inversion for terms of the form $\mathsf{inl}\,a$. In λ^{θ}, we generalized the inversion lemma to handle the fact that $\mathsf{inl}\,a$ may have an @-type as follows:

Lemma (Inversion for inl)**.** Suppose $\Gamma \vdash^{\theta} \mathsf{inl}\,a : A$. Then either

- $A = B_1 + B_2$ for some B_1 and B_2 such that $\Gamma \vdash^{\theta} a : B_1$,
- or $A = (...((B_1 + B_2)@\theta_1)...)@\theta_n$ for some B_1, B_2 and θ_1,...,θ_n such that $\Gamma \vdash^{\theta_1} a : B_1$.

An attempt to prove this lemma for LF$^{\theta}$ by induction on the typing derivation will fall apart when we reach the case for TConv:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \quad \Gamma \vdash^{\theta} a : [b_1/x]A \\ \Gamma \vdash^{\theta} [b_2/x]A : \star\end{array}}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \textsc{TConv}$$

Here, the IH will give us that $[b_1/x]A$ has one of the desired forms, but this tells us nothing about $[b_2/x]A$. The solution is to weaken the lemma so that it only claims the type given to $\mathsf{inl}\,a$ is *provably* equal to one of the desired forms, as opposed to syntactically identical. The actual inversion lemma for inl is:

Lemma 4.2.6 (Inversion for inl)**.** Suppose $\Gamma \vdash^{\theta} \mathsf{inl}\,a : A$. Then either:

- $\Gamma \vdash^{\mathsf{L}} p : A = B_1 + B_2$ for some p, B_1, and B_2 such that $\Gamma \vdash^{\theta} b : B_1$,
- or, $\Gamma \vdash^{\mathsf{L}} p : A = (...((B_1 + B_2)@\theta_1)...)@\theta_n$ for some p, B_1, B_2, and θ_1,...θ_n such that $\Gamma \vdash^{\theta_1} a : B_1$.

In the special case where A is known to have the form $B_1 + B_2$, we can show that the first case of the previous lemma applies. The proof of this makes convenient use of two unusual rules from our type system, TContra and TSumInv1, so we show it in detail.

Lemma 4.2.7 (Inversion for inl at sum types)**.** Suppose $\Gamma \vdash^\theta \mathsf{inl}\, a : B_1 + B_2$. Then $\Gamma \vdash^\theta a : B_1$.

Proof. By Lemma 4.2.6, one of two cases applies. We consider each individually:

- In the first case, we have some p, B_1' and B_2' such that there are derivations $\mathcal{D}_1 :: \Gamma \vdash^{\mathsf{L}} p : (B_1 + B_2) = (B_1' + B_2')$ and $\mathcal{D}_2 :: \Gamma \vdash^\theta b : B_1'$. But by rule TSumInv1 and $\mathcal{D}_1$, we may show $\Gamma \vdash^{\mathsf{L}} p : B_1 = B_1'$. And, thus, by rule TConv and $\mathcal{D}_2$, we have $\Gamma \vdash^\theta a : B_1$ as desired.
- In the second case, we have a proof of $\Gamma \vdash^{\mathsf{L}} p : (B_1 + B_2) = (...((B_1' + B_2')@\theta_1)...)@\theta_n$ for some p, B_1', B_2', and $\theta_1,...\theta_n$, and we know $\Gamma \vdash^{\theta_1} a : B_1'$. But the two sides of this equality have different head forms, so the desired result $\Gamma \vdash^\theta a : B_1$ will follow by TContra if we can prove that rule's well-formedness hypotheses $\Gamma \vdash^{\theta_1} B_1' : \star$ and $\Gamma \vdash^\theta B_1 : \star$. These follow from regularity and the inversion lemmas for sum and @ types.

□

Note that here we have used rule TContra explicitly to change the fragment in which a term checks rather than its type. The proof would be stuck without this use of TContra. It might be possible to continue without TContra by improving Lemma 4.2.6 to provide an explicit connection between θ and θ_1 in the second case, but it is not immediately obvious what that connection would be. In particular, any of the four combinations of θ and θ_1 is possible, so such a connection would need to include other information as well.

We can prove similar inversion lemmas for most of the other term forms in the language. The specialized versions when we know the term has a type of the expected form are useful in our proof of preservation and resemble the standard lemmas from systems without unmarked introduction forms.

The specialized versions of the inversion lemmas for functions and sigma types have a slightly different form than the one we saw for inl. In particular, they explicitly mention a substitution or value restriction, corresponding to the equality inversion typing rules for these forms.

Consider lambdas as an example. The inversion lemma itself has the same form as the corresponding lemma for inl.

Lemma 4.2.8 (Inversion for λ-expressions)**.** Suppose $\Gamma \vdash^\theta \lambda x.b : A$. Then either:

- $\Gamma \vdash^{\mathsf{L}} p : A = (x{:}B_1) \to B_2$ for some p, B_1, and B_2 such that $\Gamma, x :^\theta B_1 \vdash^\theta b : B_2$,

- or, $\Gamma \vdash^{\mathsf{L}} p : A = (...(((x : B_1) \to B_2)@\theta_1)...)@\theta_n$ for some p, B_1, B_2, and θ_1,...θ_n such that $\Gamma, x :^{\theta_1} B_1 \vdash^{\theta_1} b : B_2$.

When we specialize the lemma to the case where we know the type is an arrow type, we mention an explicit substitution into the body of the function, rather than leaving a hole for the argument:

Lemma 4.2.9 (Inversion for λ-expressions at arrow types)**.** Suppose $\Gamma \vdash^{\theta} \lambda x.b : (x : B_1) \to B_2$ and $\Gamma \vdash^{\theta} v : B_1$. Then $\Gamma \vdash^{\theta} [v/x]b : [v/x]B_2$.

The inversion lemmas for the other function forms and pairs are similar. The explicit substitution of a value for the argument to the function makes the lemma substantially easier to prove and is sufficient for preservation.

4.2.3 Preservation

The proof of preservation introduces one additional complication. We might hope to directly prove a standard statement of preservation, like this:

Theorem 4.2.10 (Preservation)**.** If $\Gamma \vdash^{\theta} a : A$ and $a \leadsto a'$, then $\Gamma \vdash^{\theta} a' : A$.

While this theorem is true for LF$^{\theta}$, an attempt to prove it directly will get stuck. Consider the case where the term is an application and the argument steps. That is, suppose $a \leadsto a'$ and we have a derivation:

$$\mathcal{D} = \frac{\Gamma \vdash^{\theta} v : (x{:}A) \to B \qquad \Gamma \vdash^{\theta} a : A \qquad \overset{\mathcal{D}'}{\Gamma \vdash^{\theta} [a/x]B : \star}}{\Gamma \vdash^{\theta} v\ a : [a/x]B}\ \textsc{TApp}$$

To complete this case of the preservation proof, we must show that $\Gamma \vdash^{\theta} v\ a' : [a/x]B$. The natural approach is to use TAPP to derive that $\Gamma \vdash^{\theta} v\ a' : [a'/x]B$, then observe that $[a/x]B$ and $[a'/x]B$ are provably equal, and conclude with TCONV.

However, there is a hitch in this plan. Our use of TAPP to conclude that $\Gamma \vdash^{\theta} v\ a' : [a'/x]B$ will require us first to show that $[a'/x]B$ is a well-formed type, i.e., that $\Gamma \vdash^{\theta} [a'/x]B : \star$. Since a' is not necessarily a value, we cannot hope to show this by inverting the well-formedness of $(x{:}A) \to B$ and using substitution. So we are stuck.

Another approach is to observe that $\Gamma \vdash^{\theta} [a'/x]B : \star$ is a potential conclusion of the IH for the subderivation $\mathcal{D}'$. Unfortunately, to use this IH we would need to show that $[a/x]B \leadsto [a'/x]B$. This may not be the case, since x might appear multiple times in B, or under binders. But this suggests a solution in the form of a more general statement of the preservation theorem itself—while it isn't the case that $[a/x]B \leadsto [a'/x]B$, it *is* the case that $[a/x]B \Rrightarrow [a'/x]B$. So if we attempt to prove preservation for parallel reductions rather than the $\leadsto$ relation, we will not be stuck here.

The following generalized statement of preservation can be proved directly by performing induction on the typing derivation and considering the possible ways for a to step in each case.

Theorem 4.2.11 (Preservation for parallel reduction). If $\Gamma \vdash^{\theta} a : A$ and $a \Rrightarrow a'$, then $\Gamma \vdash^{\theta} a' : A$.

After proving this, the original statement of preservation (Theorem 4.2.10) may be recovered as a special case, since if $a \rightsquigarrow a'$ then $a \Rrightarrow a'$.

4.3 Normalization

Our normalization proof builds upon the "partially step-indexed" technique introduced in Chapter 3. Happily, the modifications required to extend this technique to LF$^{\theta}$ are relatively standard. In particular, to properly interpret equality types we will introduce an environment to the interpretation that will keep track of values for each free variable. The rule TConv can be thought of as a *large elimination*—it changes types based on the elimination of a term. A value environment is standard for systems with large eliminations [64].

4.3.1 The Interpretation

The definition of the interpretation appears in Figure 4.7. As was the case in the previous chapter, we define two mutually recursive functions, $\mathcal{V}_{\rho}[\![A]\!]^{\theta}_{k}$ and $\mathcal{C}_{\rho}[\![A]\!]^{\theta}_{k}$. The former interprets types as sets of values and the latter interprets types as sets of expressions. The new argument ρ is the value environment. It has the form:

$$\rho ::= \emptyset \mid \rho[x \mapsto v]$$

We write $\rho\, a$ for the simultaneous substitution of values in ρ for the variables of a.

The interpretation for LF$^{\theta}$ closely mirrors the interpretation for λ^{θ}, so we will describe only the major changes. We add an interpretation for the kind $\star$, which is simply all type values. This definition suffices due to the limitations LF$^{\theta}$ imposes on the type level (as exhibited, for example, by Lemma 4.2.1). Since LF$^{\theta}$ includes three ways to introduce functions, the interpretation of arrow types is a union over the possible forms (and includes unrestricted recursion only at P). For each form, the definition follows the standard pattern of a logical relation and closely mirrors λ^{θ}, except that we extend the environment context with the argument value when interpreting the range type.

The remaining changes and additions are in the interpretation of data types—sums, pairs, and equality. In the case of sums, we add a typing assumption to maintain the property that if $v \in \mathcal{V}_{\rho}[\![A]\!]^{\theta}_{k}$, then $\cdot \vdash^{\theta} v : \rho\, A$. The interpretation for pairs is new but straightforward—a pair $\langle v_1, v_2\rangle$ is in the interpretation of $\Sigma x\!:\!A.B$ just if v_1 is in

$$
\begin{array}{lll}
\mathcal{V}_\rho[\![\star]\!]^\theta_k & = & \{v \mid \cdot \vdash^\theta v : \star\} \\
\mathcal{V}_\rho[\![\mathsf{Nat}]\!]^\theta_k & = & \{v \mid v \text{ is of the form } \mathsf{S}^n\ \mathsf{Z}\} \\
\mathcal{V}_\rho[\![A@\theta']\!]^\theta_k & = & \{v \mid v \in \mathcal{V}_\rho[\![A]\!]^{\theta'}_k\} \\
\mathcal{V}_\rho[\![(x\!:\!A) \to B]\!]^{\mathsf{L}}_k & = & \quad \{\lambda x.b \mid \cdot \vdash^{\mathsf{L}} \lambda x.b : \rho\,((x\!:\!A) \to B) \\
 & & \qquad \text{and } \forall j \le k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_j \text{ then } [v/x]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]^{\mathsf{L}}_j\} \\
 & & \cup\, \{\mathsf{ind}\, f\, x.b \mid \cdot \vdash^{\mathsf{L}} \mathsf{ind}\, f\, x.b : \rho\,((x\!:\!A) \to B) \\
 & & \qquad \text{and } \forall j \le k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_j \\
 & & \qquad\qquad \text{then } [v/x][\lambda y.\lambda z.(\mathsf{ind}\, f\, x.b)\ y/f]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]^{\mathsf{L}}_j\} \\
\mathcal{V}_\rho[\![(x\!:\!A) \to B]\!]^{\mathsf{P}}_k & = & \quad \{\lambda x.b \mid \cdot \vdash^{\mathsf{P}} \lambda x.b : \rho\,((x\!:\!A) \to B) \\
 & & \qquad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_j \text{ then } [v/x]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]^{\mathsf{P}}_j\} \\
 & & \cup\, \{\mathsf{rec}\, f\, x.b \mid \cdot \vdash^{\mathsf{P}} \mathsf{rec}\, f\, x.b : \rho\,((x\!:\!A) \to B) \\
 & & \qquad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_j \text{ then } [v/x][\mathsf{rec}\, f\, x.b/f]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]^{\mathsf{P}}_j\} \\
 & & \cup\, \{\mathsf{ind}\, f\, x.b \mid \cdot \vdash^{\mathsf{P}} \mathsf{ind}\, f\, x.b : \rho\,((x\!:\!A) \to B) \\
 & & \qquad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_j \\
 & & \qquad\qquad \text{then } [v/x][\lambda y.\lambda z.(\mathsf{ind}\, f\, x.b)\ y/f]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]^{\mathsf{P}}_j\} \\
\mathcal{V}_\rho[\![A+B]\!]^\theta_k & = & \quad \{\mathsf{inl}\, v \mid \cdot \vdash^\theta \rho\,(A+B) : \star \text{ and } v \in \mathcal{V}_\rho[\![A]\!]^\theta_k\} \\
 & & \cup\, \{\mathsf{inr}\, v \mid \cdot \vdash^\theta \rho\,(A+B) : \star \text{ and } v \in \mathcal{V}_\rho[\![B]\!]^\theta_k\} \\
\mathcal{V}_\rho[\![\Sigma x\!:\!A.B]\!]^\theta_k & = & \{\langle v_1, v_2\rangle \mid \cdot \vdash^\theta \rho\,(\Sigma x\!:\!A.B) : \star \text{ and } v_1 \in \mathcal{V}_\rho[\![A]\!]^\theta_k \text{ and } v_2 \in \mathcal{V}_{\rho[x\mapsto v_1]}[\![B]\!]^\theta_k\} \\
\mathcal{V}_\rho[\![\mu x.A]\!]^\theta_k & = & \{\mathsf{roll}\, v \mid \cdot \vdash^{\theta'} \mathsf{roll}\, v : \rho\,(\mu x.A) \text{ and } \forall j < k, v \in \mathcal{V}_\rho[\![[\mu x.A/x]A]\!]^\theta_j\} \\
\mathcal{V}_\rho[\![a_1 = a_2]\!]^\theta_k & = & \{\mathsf{refl} \mid \cdot \vdash^\theta \rho\,(a_1 = a_2) : \star \text{ and } \rho\, a_1 \Rrightarrow^* a \text{ and } \rho\, a_2 \Rrightarrow^* a \text{ for some } a\} \\
\mathcal{V}_\rho[\![A]\!]^\theta_k & = & \emptyset \qquad\qquad \text{otherwise} \\
 & & \\
\mathcal{C}_\rho[\![A]\!]^{\mathsf{P}}_k & = & \{a \mid \cdot \vdash^{\mathsf{P}} a : \rho\, A \text{ and } \forall j \le k,\ \text{if } a \leadsto^j v \text{ then } v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_{(k-j)}\} \\
\mathcal{C}_\rho[\![A]\!]^{\mathsf{L}}_k & = & \{a \mid \cdot \vdash^{\mathsf{L}} a : \rho\, A \text{ and } a \leadsto^* v \in \mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k\}
\end{array}
$$

Figure 4.7: LF$^\theta$: The Type Interpretation

the interpretation of A and v_2 is in the interpretation of $[v_1/x]B$ (though note that, to preserve the well-foundedness of the interpretation, we extend the context when interpreting B rather than substituting).

The interpretation of equality is the only case where the value environment ρ is used. The idea is simple—the interpretation of $a = b$ in the context ρ is a set containing the single term refl when $\rho\, a$ and $\rho\, b$ reduce to a common expression, and is empty otherwise. The interpretation also contains a typing assumption, for the same reason as sums.

The interpretation is a well-defined recursive function. We can formalize its descending well-founded metric as a lexicographically ordered triple (k, A, Ω), as we did in Chapter 3. Here, k is the index, A is the expression and Ω is one of $\mathcal{C}$ or $\mathcal{V}$ with $\mathcal{V} < \mathcal{C}$. The third element of the triple tracks which interpretation is being called—the computational interpretation may call the value interpretation at the same index and type, but not vice-versa.

4.3.2 The Proof

The proof of normalization using the interpretation above closely resembles the proof from Chapter 3. The primary difference is additional infrastructure to deal with the value environment. In particular, we begin by defining the judgement $\rho \models_k \Gamma$, which indicates that ρ is a good model for the context Γ, for at least k steps.

$$\frac{}{\emptyset \models_k \cdot}\ \textsc{ENil} \qquad \frac{\rho \models_k \Gamma \quad v \in \mathcal{V}_\rho[\![A]\!]^\theta_k \quad \Gamma \vdash^\theta A : \star}{\rho[x \mapsto v] \models_k \Gamma, x :^\theta A}\ \textsc{ECons}$$

Intuitively, $\rho \models_k \Gamma$ asserts that ρ maps term variables to well-behaved values. Because of the premise $\Gamma \vdash^\theta A : \star$, it also asserts that Γ does not contain any type variables. This is vacuously true for the empty context and preserved by each case of the type interpretation.

The soundness theorem relies on a few key lemmas about the interpretation. These lemmas may all be proved by induction over the same ordering which we used to demonstrate that the interpretation is well-defined. The first is a standard "downward closure" property for step-indexed relations: it says that requiring values to stay well-behaved for a larger number of steps creates a more precise interpretation.

Lemma 4.3.1. For any A, θ and ρ, if $j \leq k$ then $\mathcal{V}_\rho[\![A]\!]^\theta_k \subseteq \mathcal{V}_\rho[\![A]\!]^\theta_j$.

The next two lemmas relate the L and P interpretations of a type. They are used to handle the TSub and TMVal rules, respectively. The first says that the set of logical values is a subset of the corresponding programmatic sets. Recall that Ω ranges over $\mathcal{V}$ and $\mathcal{C}$.

Lemma 4.3.2. For any A, k, θ and ρ, $\Omega_\rho[\![A]\!]^{\mathsf{L}}_k \subseteq \Omega_\rho[\![A]\!]^{\mathsf{P}}_k$.

The second says that for mobile types, the reverse containment also holds. For these types, the interpretations contain the same values in both fragments.

Lemma 4.3.3. For any k and ρ, if $\mathsf{Mob}\,(A)$ then $\mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k \subseteq \mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k$.

Finally, because of the TConv rule, we need equal types to have the same interpretation.

Lemma 4.3.4. Suppose $\rho\, B_1 \Rrightarrow^* A$ and $\rho\, B_2 \Rrightarrow^* A$ and $\Gamma \vdash^\theta B_1 : \star$ and $\Gamma \vdash^\theta B_2 : \star$ and $\rho \models_k \Gamma$. Then $a \in \Omega_\rho[\![B_1]\!]^\theta_k$ iff $a \in \Omega_\rho[\![B_2]\!]^\theta_k$.

We can now prove the main soundness result by induction on $\Gamma \vdash^\theta a : A$. Normalization is an immediate corollary. We also get a characterization of which terms can be proven equal in the empty context. We need such a characterization to prove progress.

Theorem 4.3.5 (Soundness)**.** If $\Gamma \vdash^\theta a : A$ and $\rho \models_k \Gamma$, then $\rho\, a \in \mathcal{C}_\rho[\![A]\!]^\theta_k$.

Corollary 4.3.6 (Normalization)**.**
If $\cdot \vdash^{\mathsf{L}} a : A$, then there exists a value v such that $a \leadsto^* v$.

Corollary 4.3.7 (Soundness of propositional equality)**.**
If $\cdot \vdash^{\mathsf{L}} a : A_1 = A_2$, then there exists some A such that $A_1 \Rrightarrow^* A$ and $A_2 \Rrightarrow^* A$.

Normalization holds only for closed terms. This is a result of the fact that uses of the TConv rule are unmarked in the syntax. It is possible to assume a contradictory equality and use it to typecheck a non-terminating term in the logical fragment. For example, the following statement is derivable:

$$y :^{\mathsf{L}} \mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat}) \vdash^{\mathsf{L}} (\lambda x.x\ x)\ (\lambda x.x\ x) : \mathsf{Nat}$$

This distinguishes LF$^\theta$ from intensional type theories like Coq and Agda. In those systems, our rule TConv arises as the pattern-matching elimination form for a defined equality datatype. Uses of this eliminator would appear in the term above, and their reduction would get "stuck" on the variable y, since it does not reduce to the appropriate constructor.

The benefit of giving up normalization of open terms is a more generous equality. Since uses of conversion appear in terms in Coq and Agda, they often get in the way of judging two terms which use such conversions equal. In our system, this cannot happen. The drawback is that the typechecker cannot automatically normalize expressions (since they may diverge), so, in a surface language, some uses of refl must be explicit and annotated with a maximum step count. However, in a language with general recursion some explicit proofs are unavoidable, since checking a logical term can involve reducing a programmatic term that appears in its type. Since our language must accommodate such proofs in any case, making conversion unmarked is appealing.

4.4 Progress

The progress theorem relies on canonical forms lemmas, which we may now prove. As described in Section 4.2.1, the TCONV and TCONTRA cases of each proof will require us to rule out inconsistent equalities such as $(\mathsf{Nat} + \mathsf{Nat}) = \mathsf{Nat}$. Therefore, these lemmas rely on Corollary 4.3.7. Otherwise, the proofs go by the technique described in Section 3.2.1.

Lemma 4.4.1 (Canonical forms for Nat)**.** If $\cdot \vdash^{\theta} v : \mathsf{Nat}$ then either $v = \mathsf{Z}$ or $v = \mathsf{S}\ v'$ for some value v'.

Lemma 4.4.2 (Canonical forms for arrows)**.** If $\cdot \vdash^{\theta} v : (x : A) \to B$ then either $v = (\lambda x.b)$, $v = \mathsf{rec}\, f\ x.b$, or $v = \mathsf{ind}\, f\ x.b$ for some f and b.

Lemma 4.4.3 (Canonical forms for sums)**.** If $\cdot \vdash^{\theta} v : B_1 + B_2$ then there is some v' such that $v = \mathsf{inl}\, v'$ or $v = \mathsf{inr}\, v'$.

Lemma 4.4.4 (Canonical forms for pairs)**.** If $\cdot \vdash^{\theta} v : \Sigma x : B_1.B_2$ then there are some v_1 and v_2 such that $v = \langle v_1, v_2 \rangle$.

Lemma 4.4.5 (Canonical forms for recursive types)**.** If $\cdot \vdash^{\theta} v : \mu alpha.A$ then $v = \mathsf{roll}\, b'$ for some v'.

The progress theorem is then an easy induction on $\cdot \vdash^{\theta} a : A$.

Theorem 4.4.6 (Progress)**.** If $\cdot \vdash^{\theta} a : A$, then either a is a value or there exists a' such that $a \rightsquigarrow a'$.

Chapter 5

Adding Polymorphism and Type-Level Computation

I know what my weaknesses are, probably better than you do.
Revolutionary Chinese propaganda, the color of blue.
I thought I knew what my weaknesses were anyway,
Then the orange tree blossomed last Saturday.
There was nothing in it but pain for me.

Ontario
The Mountain Goats

This chapter introduces the language PCC$^\theta$, which extends LF$^\theta$ with predicative polymorphism and type-level computation. Thus, PCC$^\theta$ is a complete dependently typed core language in its own right. We have chosen the name PCC$^\theta$ to emphasize that it can be thought of as a predicative variant of the Calculus of Constructions, enhanced with the novel logicality and equality features described above. The primary contribution of the chapter is a proof that this language is type safe and that its logical fragment is normalizing and consequently consistent as a logic. This demonstrates that the novel features we have introduced can soundly coexist with a "full-spectrum" dependent type theory.

On the other hand, a number of features we included for convenience in THETA are absent from PCC$^\theta$. Some of these, such as a general notion of datatypes and a terminating recursor, have been omitted because we believe they are not likely to introduce genuine problems, but would introduce many complex cases to an already very long proof. Others, like the universe hierarchy, have been omitted because their metatheory has proved intractable in the setting of LF$^\theta$. Omissions of the latter kind are considered in detail in Chapter 6.

Unlike the proofs in Chapters 3 and 4, the metatheory of PCC$^\theta$ has not been mechanized. Type-level computation, in particular, substantially complicates the definition of the interpretation of types used in the proof of normalization of the logical

Sorts

$s \quad ::= \quad \star_\tau \mid \star_\sigma$

Kinds

$k \quad ::= \quad s \mid (x : A) \to k \mid (x : k_1) \to k_2$

Types

$A,\ B \quad ::= \quad x \mid (x : A) \to B \mid (x : k) \to B \mid \lambda x : A.B \mid \lambda x : k.B \mid B\ a \mid B\ A$
$\quad \mid \mathsf{Nat} \mid A + B \mid \Sigma x\!:\!A\,.\,B \mid \mu\,x.A \mid A@\theta \mid a = b \mid A = B \mid$

Terms

$a,\ b \quad ::= \quad x \mid \lambda x\!:\!A.b \mid \lambda x\!:\!k.b \mid \mathsf{rec}\, f\,(x\!:\!A).b \mid \mathsf{rec}\, f\,(x\!:\!k).b \mid b\ a \mid b\ A$
$\quad \mid \mathsf{Z} \mid \mathsf{S}\, a \mid \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow b_1; \mathsf{S}\ x \Rightarrow b_2\} \mid$
$\quad \mid \mathsf{inl}\, a \mid \mathsf{inr}\, a \mid \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1; \mathsf{inr}\ y \Rightarrow b_2\} \mid$
$\quad \mid \langle a, b\rangle \mid \mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b \mid \mathsf{roll}\, a \mid \mathsf{unroll}\, a \mid \mathsf{refl} \mid \mathsf{trefl}$

Consistency Classifiers

$\theta \quad ::= \quad \mathsf{L} \mid \mathsf{P}$

Type Values

$V \quad ::= \quad x \mid (x : A) \to B \mid (x : k) \to B \mid \lambda x : A.B \mid \lambda x : k.B$
$\quad \mid \mathsf{Nat} \mid A + B \mid \Sigma x\!:\!A\,.\,B \mid \mu\,x.A \mid A@\theta \mid a = b \mid A = B \mid$

Term Values

$v \quad ::= \quad x \mid \lambda x\!:\!A.b \mid \lambda x\!:\!k.b \mid \mathsf{rec}\, f\,(x\!:\!A).b \mid \mathsf{rec}\, f\,(x\!:\!k).b$
$\quad \mid \mathsf{Z} \mid \mathsf{S}\, v \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \langle v_1, v_2\rangle \mid \mathsf{roll}\, v \mid \mathsf{refl} \mid \mathsf{trefl}$

Figure 5.1: PCC$^\theta$: Syntax

fragment. Where previous interpretations always computed a set of expressions, the interpretation we define in Section 5.4 can produce several different kinds of set-theoretic objects and is therefore substantially more work to model in a proof assistant like Coq. For this reason, the metatheory of PCC$^\theta$ appears here in great detail.

5.1 The PCC$^\theta$ Language

Since PCC$^\theta$ is quite similar to both LF$^\theta$ and Theta which have been described earlier in this thesis, we will focus here only on the differences. The syntax of PCC$^\theta$ appears in Figure 5.1. Unlike LF$^\theta$ and Theta, we present PCC$^\theta$ with a stratified syntax—terms, types, and kinds are their own syntactic categories. This introduces some unfortunate duplication in the typing and reduction relations, but is necessary due

$$\boxed{a \rightsquigarrow b}$$

$$\frac{}{(\lambda x\!:\!A.b)\ v \rightsquigarrow [v/x]b}\ \textsc{SBeta} \qquad \frac{}{(\lambda x\!:\!k.b)\ V \rightsquigarrow [V/x]b}\ \textsc{SBetaT}$$

$$\frac{}{(\mathsf{rec}\, f\ (x\!:\!A).b)\ v \rightsquigarrow [v/x][\mathsf{rec}\, f\ (x\!:\!A).b/f]b}\ \textsc{SFBeta}$$

$$\frac{}{(\mathsf{rec}\, f\ (x\!:\!k).b)\ V \rightsquigarrow [V/x][\mathsf{rec}\, f\ (x\!:\!k).b/f]b}\ \textsc{SFBetaT}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{Z}\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z]a_1}\ \textsc{SCaseZ}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{S}\, v\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_2}\ \textsc{SCaseS}$$

$$\boxed{A \rightsquigarrow B}$$

$$\frac{}{(\lambda x : A.B)\ v \rightsquigarrow [v/x]B}\ \textsc{TSBeta} \qquad \frac{}{(\lambda x : k.B)\ V \rightsquigarrow [V/x]B}\ \textsc{TSBetaT}$$

Figure 5.2: PCC$^\theta$: Operational Semantics (excerpt)

to metatheoretic complications described in Section 6.4. The only unusual pieces of syntax are the kinds $\star_\tau$ and $\star_\sigma$, which are used to handle predicative polymorphism and are described below.

The operational semantics of PCC$^\theta$ appear in Figure 5.2. For convenience in the proof below, we have written out congruence rules for the reduction relations rather then specifying them via evaluation contexts. For readability, we do not show the complete definition here, but it may be found in Appendix A. The definition of parallel reduction, which is used in the typing rules for equality proofs and in the metatheory, may also be found in Appendix A.

The typing rules for functions, function types, variables and sorts appear in Figure 5.3. There are four mutually defined relations—$\Gamma \vdash^\theta a : A$ assigns types to terms, $\Gamma \vdash A : k$ assigns kinds to types, $\Gamma \vdash k$ checks kinds, and $\vdash \Gamma$ checks contexts.

Since the system uses a stratified syntax, we have four distinct arrow forms (two types and two kinds). We have given the typing rules names reflecting a correspondence with the four "rules" in the pure type system presentation of the Calculus of Constructions [6]. In particular, KArrTLC handles type-level computation (CC rule $(\Box, \Box)$), KArrDep handles dependent lambdas (CC rule $(\star, \Box)$), and TArrComp handles normal term-level computation (CC rule $(\star, \star)$). On the other hand, TArrPoly handles polymorphism but does not correspond directly to the CC rule $(\Box, \star)$ because PCC$^\theta$ is predicative, as we will see shortly.

The system also includes recursive function forms at the term level, but not at the

$$\boxed{\Gamma \vdash k}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star_\tau}\ \textsc{KSort} \qquad \frac{\Gamma \vdash k_1 \quad \Gamma, x : k_1 \vdash k_2}{\Gamma \vdash (x : k_1) \to k_2}\ \textsc{KArrTLC} \qquad \frac{\begin{array}{c}\Gamma \vdash A : \star_\sigma \quad \mathsf{Mob}\,(A) \\ \Gamma, x : A \vdash k\end{array}}{\Gamma \vdash (x : A) \to k}\ \textsc{KArrDep}$$

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

$$\frac{(x : k) \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : k}\ \textsc{TVar} \qquad \frac{\Gamma \vdash A : \star_\tau}{\Gamma \vdash A : \star_\sigma}\ \textsc{TMonoPoly}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : s \quad \mathsf{Mob}\,(A) \\ \Gamma, x : A \vdash B : s\end{array}}{\Gamma \vdash (x : A) \to B : s}\ \textsc{TArrComp} \qquad \frac{\begin{array}{c}\Gamma \vdash k \\ \Gamma, x : k \vdash B : \star_\sigma\end{array}}{\Gamma \vdash (x : k) \to B : \star_\sigma}\ \textsc{TArrPoly}$$

$$\frac{\begin{array}{c}\Gamma, x : k_1 \vdash B : k_2 \\ \Gamma \vdash (x : k_1) \to k_2\end{array}}{\Gamma \vdash \lambda x : k_1.B : (x : k_1) \to k_2}\ \textsc{TLamTLC} \qquad \frac{\begin{array}{c}\Gamma, x : A \vdash B : k_2 \\ \Gamma \vdash (x : A) \to k_2\end{array}}{\Gamma \vdash \lambda x : A.B : (x : A) \to k_2}\ \textsc{TLamDep}$$

$$\frac{\begin{array}{c}\Gamma \vdash B : (x : k_1) \to k_2 \\ \Gamma \vdash A : k_1 \quad \Gamma \vdash [A/x]k_2\end{array}}{\Gamma \vdash B\ A : [A/x]k_2}\ \textsc{TAppTLC} \qquad \frac{\begin{array}{c}\Gamma \vdash B : (x : A) \to k \\ \Gamma \vdash^{\mathsf{L}} a : A \quad \Gamma \vdash [a/x]k\end{array}}{\Gamma \vdash B\ a : [a/x]k}\ \textsc{TAppDep}$$

$$\boxed{\Gamma \vdash^\theta a : A}$$

$$\frac{(x : A) \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash^\theta x : A}\ \textsc{EVar}$$

$$\frac{\begin{array}{c}\Gamma, x : A \vdash^{\mathsf{L}} b : B \\ \Gamma \vdash (x : A) \to B : \star_\sigma\end{array}}{\Gamma \vdash^{\mathsf{L}} \lambda x{:}A.b : (x : A) \to B}\ \textsc{ELamComp} \qquad \frac{\begin{array}{c}\Gamma, x : k \vdash^{\mathsf{L}} b : B \\ \Gamma \vdash (x : k) \to B : \star_\sigma\end{array}}{\Gamma \vdash^{\mathsf{L}} \lambda x{:}k.b : (x : k) \to B}\ \textsc{ELamPoly}$$

$$\frac{\begin{array}{l}\Gamma, f : (x : A) \to B, x : A \vdash^{\mathsf{P}} b : B \\ \Gamma \vdash (x : A) \to B : \star_\sigma\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\,(x{:}A).a : (x : A) \to B}\ \textsc{ERecComp} \qquad \frac{\begin{array}{l}\Gamma, f : (x : k) \to B, x : k \vdash^{\mathsf{P}} b : B \\ \Gamma \vdash (x : k) \to B : \star_\sigma\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\,(x{:}k).a : (x : k) \to B}\ \textsc{ERecPoly}$$

$$\frac{\begin{array}{l}\Gamma \vdash^\theta b : (x : A) \to B \\ \Gamma \vdash^\theta a : A \quad \Gamma \vdash [a/x]B : \star_\sigma\end{array}}{\Gamma \vdash^\theta b\ a : [a/x]B}\ \textsc{EAppComp} \qquad \frac{\begin{array}{l}\Gamma \vdash^\theta b : (x : k) \to B \\ \Gamma \vdash A : k \quad \Gamma \vdash [A/x]B : \star_\sigma\end{array}}{\Gamma \vdash^\theta b\ A : [A/x]B}\ \textsc{EAppPoly}$$

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

$$\frac{}{\vdash \cdot}\ \textsc{CNil} \qquad \frac{\begin{array}{c}\vdash \Gamma \quad \Gamma \vdash A : \star_\sigma \\ \mathsf{Mob}\,(A) \quad x \notin \mathsf{dom}(\Gamma)\end{array}}{\vdash \Gamma, x : A}\ \textsc{CTrm} \qquad \frac{\begin{array}{c}\vdash \Gamma \quad \Gamma \vdash k \\ x \notin \mathsf{dom}(\Gamma)\end{array}}{\vdash \Gamma, x : k}\ \textsc{CTyp}$$

Figure 5.3: PCC$^\theta$ Typing: Basics

type-level. This mirrors THETA's restriction that types must check logically, which is discussed in greater detail in Section 6.1. Restricting the type level to "logical" constructs permits a few other simplifications in the specification of PCC$^\theta$. The kinding judgement $\Gamma \vdash A : k$ is not indexed by a logicality (though one may imagine an implicit logicality L when comparing to the other systems in this document). Similarly, we do not mark type variables with a logicality in the context. In fact, the typing rules for terms can be stated in such a way that only mobile types are assigned to term variables in the context, so the stratified syntax allows us to avoid marking any variables at all with logicalities, as we see in the definition of $\vdash \Gamma$.

In order to enforce predicative polymorphism, we use two sorts for types: $\star_\sigma$ and $\star_\tau$. If $\Gamma \vdash A : \star_\tau$, then we know A is a monomorphic type. On the other hand, $\star_\sigma$ may classify any type, including those that make use of polymorphism. The names of these sorts are intended to evoke systems like Standard ML, which syntactically distinguishes between monotypes τ and type schemes σ:

$$\begin{array}{lcl} \sigma & ::= & \tau \mid \forall\alpha.\sigma \\ \tau & ::= & \alpha \mid \mathsf{Nat} \mid \tau \to \tau \mid \tau_1 + \tau_2 \mid \ \ldots \end{array}$$

In PCC$^\theta$, the σs are more general because they are not required to be in prenex form. Our types look more like the following grammar.

$$\begin{array}{lcl} \sigma & ::= & \tau \mid \forall\alpha.\sigma \mid \sigma \to \sigma \mid \sigma_1 + \sigma_2 \mid \ \ldots \\ \tau & ::= & \alpha \mid \mathsf{Nat} \mid \tau \to \tau \mid \tau_1 + \tau_2 \mid \ \ldots \end{array}$$

However, type variables still only range over monotypes τ; one can never instantiate a $\forall$-type with another polymorphic type. We have chosen to use sorts to distinguish polytypes from monotypes rather than separating the syntax because it avoids duplication in the typing rules and because this allows us to demonstrate the interesting result that, while a completely collapsed syntax causes problems for our proof techniques (Section 6.4), we can at least collapse this much.

The predicativity restriction is captured in the formalism by the rule TARRPOLY and the judgement $\Gamma \vdash k$. In particular, for k to be a valid domain for a polymorphic arrow type, it must be the case that $\Gamma \vdash k$. It is not the case that $\Gamma \vdash \star_\sigma$. Polymorphic types are themselves given the kind $\star_\sigma$, so the domain of a polymorphic arrow type never ranges over itself. Thus, the judgement $\Gamma \vdash k$ can be read as "k is a kind that classifies monotypes", and there are valid kinds for which $\Gamma \vdash k$ does not hold (e.g., $\star_\sigma$). The rule TMONOPOLY allows monotypes to be used in contexts where polytypes are expected (just as τ is included in σ in the grammars above). In particular, this means that we can use $\Gamma \vdash A : \star_\sigma$ as a general well-formedness check on A when we do not care whether or not it is polymorphic.

Like THETA, λ^θ and LF$^\theta$, PCC$^\theta$ allows the two fragments to interact in a variety of ways, as shown in Figure 5.4. In particular, PCC$^\theta$ can internalize its typing judgement via the $A@\theta$ type form, and data computed by the programmatic fragment can be

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

$$\frac{\Gamma \vdash A : s}{\Gamma \vdash A@\theta : s}\ \textsc{TAt}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\theta} v : A@\theta' \\ \Gamma \vdash A : \star_{\sigma}\end{array}}{\Gamma \vdash^{\theta'} v : A}\ \textsc{EUnboxVal} \qquad \frac{\begin{array}{c}\mathsf{Mob}\,(A) \\ \Gamma \vdash^{\mathsf{P}} v : A\end{array}}{\Gamma \vdash^{\mathsf{L}} v : A}\ \textsc{EMVal}$$

$$\frac{\Gamma \vdash^{\theta} a : A}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}\ \textsc{EBoxP} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{L}} a : A@\theta}\ \textsc{EBoxL} \qquad \frac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\ \textsc{EBoxLV}$$

$$\boxed{\mathsf{Mob}\,(A)}$$

$$\frac{}{\mathsf{Mob}\,(\mathsf{Nat})}\ \textsc{MNat} \qquad \frac{}{\mathsf{Mob}\,(a = b)}\ \textsc{MEq} \qquad \frac{}{\mathsf{Mob}\,(A = B)}\ \textsc{MEqT}$$

$$\frac{}{\mathsf{Mob}\,(A@\theta)}\ \textsc{MAt} \qquad \frac{\mathsf{Mob}\,(A) \quad \mathsf{Mob}\,(B)}{\mathsf{Mob}\,(A + B)}\ \textsc{MSum} \qquad \frac{\mathsf{Mob}\,(A) \quad \mathsf{Mob}\,(B)}{\mathsf{Mob}\,(\Sigma x{:}A\,.\,B)}\ \textsc{MSigma}$$

Figure 5.4: PCC$^{\theta}$ Typing: The Fragments

used in the logical fragment via the EMVal rule. We do not need an equivalent kind form, as in $k@\theta$, since the kinding judgement is not indexed by a fragment.

The rules for datatypes appear in Figure 5.5. One small difference from previous systems appears in the pattern matching rules for sums and products, where we tag the potentially non-mobile types of the scrutinee's components with an explicit @-type, to preserve the property that term variables in the context are assigned mobile types.

Finally, the rules for equality appear in Figure 5.6. Due to the stratified syntax, there are two equality forms in PCC$^{\theta}$—one for terms and one for types. Both behave similarly to LF$^{\theta}$'s equality, and both may be used for conversion during typechecking and kind checking (rules EConv, EConvT, TConv and TConvT). As in LF$^{\theta}$, the inversion lemmas for our preservation proof require us to add injectivity axioms for the system's type constructors. They appear in Figure 5.7.

5.2 Syntactic Metatheory

In this section, we will prove basic syntactic results for PCC$^{\theta}$. The goal of this chapter is to demonstrate that the logical fragment of PCC$^{\theta}$ is consistent, and to verify this

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat} : \star_\tau}\ \textsc{TNat} \qquad \frac{\begin{array}{c}\Gamma \vdash A : s\\ \Gamma \vdash B : s\end{array}}{\Gamma \vdash A + B : s}\ \textsc{TSum}$$

$$\frac{\begin{array}{l}\Gamma \vdash A : s \quad \mathsf{Mob}\,(A)\\ \Gamma, x : A \vdash B : s\end{array}}{\Gamma \vdash \Sigma x\!:\!A\,.\,B : s}\ \textsc{TSigma} \qquad \frac{\Gamma, x : \star_\tau \vdash A : \star_\tau}{\Gamma \vdash \mu\, x.A : \star_\tau}\ \textsc{TMu}$$

$$\boxed{\Gamma \vdash^\theta a : A}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Z} : \mathsf{Nat}}\ \textsc{EZero} \qquad \frac{\Gamma \vdash^\theta a : \mathsf{Nat}}{\Gamma \vdash^\theta \mathsf{S}\, a : \mathsf{Nat}}\ \textsc{ESucc}$$

$$\frac{\begin{array}{l}\Gamma \vdash^\theta a : \mathsf{Nat} \quad \Gamma \vdash B : \star_\sigma\\ \Gamma, z : \mathsf{Z} = a \vdash^\theta b_1 : B\\ \Gamma, x : \mathsf{Nat}, z : (\mathsf{S}\, x) = a \vdash^\theta b_2 : B\end{array}}{\Gamma \vdash^\theta \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow b_1; \mathsf{S}\ x \Rightarrow b_2\} : B}\ \textsc{ENCase}$$

$$\frac{\begin{array}{l}\Gamma \vdash^\theta a : A\\ \Gamma \vdash A + B : \star_\sigma\end{array}}{\Gamma \vdash^\theta \mathsf{inl}\, a : A + B}\ \textsc{EInl} \qquad \frac{\begin{array}{l}\Gamma \vdash^\theta b : B\\ \Gamma \vdash A + B : \star_\sigma\end{array}}{\Gamma \vdash^\theta \mathsf{inr}\, b : A + B}\ \textsc{EInr}$$

$$\frac{\begin{array}{l}\Gamma \vdash^\theta a : (A_1 + A_2)@\theta' \quad \Gamma \vdash B : \star_\sigma\\ \Gamma, x : A_1@\theta', z : \mathsf{inl}\, x = a \vdash^\theta b_1 : B\\ \Gamma, x : A_2@\theta', z : \mathsf{inr}\, x = a \vdash^\theta b_2 : B\end{array}}{\Gamma \vdash^\theta \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1; \mathsf{inr}\ x \Rightarrow b_2\} : B}\ \textsc{ESCase}$$

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma x\!:\!A\,.\,B : \star_\sigma\\ \Gamma \vdash^\theta a : A\\ \Gamma \vdash^\theta b : [a/x]B\end{array}}{\Gamma \vdash^\theta \langle a, b\rangle : \Sigma x\!:\!A\,.\,B}\ \textsc{EPair} \qquad \frac{\begin{array}{l}\Gamma \vdash^\theta a : (\Sigma x\!:\!A_1\,.\,A_2)@\theta'\\ \Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a \vdash^\theta b : B\\ \Gamma \vdash B : \star_\sigma\end{array}}{\Gamma \vdash^\theta \mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b : B}\ \textsc{EPCase}$$

$$\frac{\begin{array}{l}\Gamma \vdash^\theta a : [\mu\, x.A/x]A\\ \Gamma \vdash \mu\, x.A : \star_\sigma\end{array}}{\Gamma \vdash^\theta \mathsf{roll}\, a : \mu\, x.A}\ \textsc{ERoll} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{P}} a : \mu\, x.A\\ \Gamma \vdash [\mu\, x.A/x]A : \star_\sigma\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{unroll}\, a : [\mu\, x.A/x]A}\ \textsc{EUnroll}$$

Figure 5.5: PCC$^\theta$ Typing: Datatypes

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

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \quad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash a = b : \star_\tau}\ \text{TEq} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \quad \Gamma \vdash A : [b_1/x]k \\ \Gamma \vdash [b_2/x]k\end{array}}{\Gamma \vdash A : [b_2/x]k}\ \text{TConv}$$

$$\frac{\Gamma \vdash A : k_1 \quad \Gamma \vdash B : k_2}{\Gamma \vdash A = B : \star_\tau}\ \text{TEqT} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} b : B_1 = B_2 \quad \Gamma \vdash A : [B_1/x]k \\ \Gamma \vdash [B_2/x]k\end{array}}{\Gamma \vdash A : [B_2/x]k}\ \text{TConvT}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{l}a \Rrightarrow^* c \quad b \Rrightarrow^* c \\ \Gamma \vdash^{\theta_1} a : A \quad \Gamma \vdash^{\theta_2} b : B\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = b}\ \text{ERefl} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \quad \Gamma \vdash^{\theta} a : [b_1/x]A \\ \Gamma \vdash [b_2/x]A : \star_\sigma\end{array}}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \text{EConv}$$

$$\frac{\begin{array}{l}A \Rrightarrow^* C \quad B \Rrightarrow^* C \\ \Gamma \vdash A : k_1 \quad \Gamma \vdash B : k_2\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{trefl} : A = B}\ \text{EReflT} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} b : B_1 = B_2 \quad \Gamma \vdash^{\theta} a : [B_1/x]A \\ \Gamma \vdash [B_2/x]A : \star_\sigma\end{array}}{\Gamma \vdash^{\theta} a : [B_2/x]A}\ \text{EConvT}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a_1 : B_1 = B_2 \quad \Gamma \vdash^{\theta} a : A \\ \mathsf{hd}\,(B_1) = \mathit{hf}_1 \quad \mathsf{hd}\,(B_2) = \mathit{hf}_2 \quad \mathit{hf}_1 \neq \mathit{hf}_2 \\ \Gamma \vdash A : \star_\sigma \quad \Gamma \vdash B : \star_\sigma\end{array}}{\Gamma \vdash^{\theta'} a : B}\ \text{EContra}$$

Head Forms

$\mathit{hf} \quad ::= \quad \mathsf{HArrComp} \mid \mathsf{HArrPoly} \mid \mathsf{HAt}\,\theta \mid \mathsf{HNat} \mid \mathsf{HSum} \mid \mathsf{HSigma} \mid \mathsf{HEq} \mid \mathsf{HEqT} \mid \mathsf{HMu}$

$$\boxed{\mathsf{hd}\,(A) = \mathit{hf}}$$

$$\frac{}{\mathsf{hd}\,((x : A) \to B) = \mathsf{HArrComp}}\ \text{HArrComp} \qquad \frac{}{\mathsf{hd}\,((x : k) \to B) = \mathsf{HArrPoly}}\ \text{HArrPoly}$$

$$\frac{}{\mathsf{hd}\,(A@\theta) = \mathsf{HAt}\,\theta}\ \text{HAt} \qquad \frac{}{\mathsf{hd}\,(\Sigma x{:}A\,.\,B) = \mathsf{HSigma}}\ \text{HSigma}$$

$$\frac{}{\mathsf{hd}\,(\mathsf{Nat}) = \mathsf{HNat}}\ \text{HNat} \qquad \frac{}{\mathsf{hd}\,(A + B) = \mathsf{HSum}}\ \text{HSum}$$

$$\frac{}{\mathsf{hd}\,(\mu\, x.A) = \mathsf{HMu}}\ \text{HMu} \qquad \frac{}{\mathsf{hd}\,(a = b) = \mathsf{HEq}}\ \text{HEq} \qquad \frac{}{\mathsf{hd}\,(A = B) = \mathsf{HEqT}}\ \text{HEqT}$$

Figure 5.6: PCC$^\theta$ Typing: Equality

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (A@\theta') = (B@\theta') \\ \Gamma \vdash A = B : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : A = B}\ \textsc{EAtInv}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : ((x : A_1) \to A_2) = ((x : B_1) \to B_2) \\ \Gamma \vdash A_1 = B_1 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : A_1 = B_1}\ \textsc{EArrInv1}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : ((x : k_1) \to A_2) = ((x : k_2) \to B_2) \\ \Gamma \vdash V : k_1 \\ \Gamma \vdash [V/x]A_2 = [V/x]B_2 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : [V/x]A_2 = [V/x]B_2}\ \textsc{EArrInvT2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : ((x : A_1) \to A_2) = ((x : B_1) \to B_2) \\ \Gamma \vdash^{\theta'} v : A_1 \\ \Gamma \vdash [v/x]A_2 = [v/x]B_2 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : [v/x]A_2 = [v/x]B_2}\ \textsc{EArrInv2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\Sigma x{:}A_1 \,.\, A_2) = (\Sigma x{:}B_1 \,.\, B_2) \\ \Gamma \vdash A_1 = B_1 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : A_1 = B_1}\ \textsc{ESigmaInv1}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\Sigma x{:}A_1 \,.\, A_2) = (\Sigma x{:}B_1 \,.\, B_2) \\ \Gamma \vdash^{\theta'} v : A_1 \\ \Gamma \vdash [v/x]A_2 = [v/x]B_2 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : [v/x]A_2 = [v/x]B_2}\ \textsc{ESigmaInv2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (A_1 + A_2) = (B_1 + B_2) \\ \Gamma \vdash A_1 = B_1 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : A_1 = B_1}\ \textsc{ESumInv1}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (A_1 + A_2) = (B_1 + B_2) \\ \Gamma \vdash A_2 = B_2 : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : A_2 = B_2}\ \textsc{ESumInv2}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\mu\, x.A) = (\mu\, x.B) \\ \Gamma \vdash ([\mu\, x.A/x]A) = ([\mu\, x.B/x]B) : \star_{\tau}\end{array}}{\Gamma \vdash^{\theta} a : ([\mu\, x.A/x]A) = ([\mu\, x.B/x]B)}\ \textsc{EMuInv}$$

Figure 5.7: PCC$^{\theta}$ Typing: Injectivity Axioms

result in detail. For this reason, we have written explicit proofs for many of the basic lemmas below. As we will see in Chapter 6, the unmarked equality in PCC$^\theta$ can cause surprising problems, so it is important to explicitly check as much as possible. Despite this, many of the early lemmas will be stated without proof, both because the proofs are uninteresting and because writing out proofs for everything is infeasible.[1]

5.2.1 Reduction Basics

We begin with simple results about the reduction relations for PCC$^\theta$. These lemmas are all proved by a straightforward structural induction, so we omit the details. In principle, dozens of lemmas about reduction are needed for the proofs in later sections. For readability, we have stated only the lemmas that are actually used in proofs that appear in detail below.

Lemma 5.2.1 (Parallel reduction inversion for @)**.** If $A@\theta \Rrightarrow^* B@\theta$ then $A \Rrightarrow^* B$.

Lemma 5.2.2 (Parallel reduction inversion for arrows)**.** If $(x : A_1) \to A_2 \Rrightarrow^* (x : B_1) \to B_2$ then $A_1 \Rrightarrow^* B_1$ and $A_2 \Rrightarrow^* B_2$.

Lemma 5.2.3 (Application evaluation inversion)**.** Suppose $b\ a \leadsto^j v$. Then there exist $i_1, i_2, i_3 \in \mathbb{N}$ such that $1 + i_1 + i_2 + i_3 = j$ and $a \leadsto^{i_2} v'$ and either

- $b \leadsto^{i_1} \lambda x\!:\!A.b'$ for some b' such that $[v'/x]b' \leadsto^{i_3} v$,
- or $b \leadsto^{i_1} \mathsf{rec}\, f\ (x\!:\!A).b'$ for some b' such that $[v'/x][\mathsf{rec}\, f\ (x\!:\!A).b'/f]b' \leadsto^{i_3} v$.

Lemma 5.2.4 (pcase evaluation inversion)**.** Suppose $\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle \Rightarrow b \leadsto^j v$. Then there exist $i_1, i_2 \in \mathbb{N}$ such that $1 + i_1 + i_2 = j$ and $a \leadsto^{i_1} \langle v_1, v_2\rangle$ and $[v_1/x][v_2/y][\mathsf{refl}/z]b \leadsto^{i_2} v$.

Lemma 5.2.5 (Term substitution preserves head forms)**.** If $\mathsf{hd}\,(A) = hf$, then for any v and x, $\mathsf{hd}\,([v/x]A) = hf$.

Lemma 5.2.6. For any v, x and v', $[v'/x]v$ is a value.

Lemma 5.2.7 (Term substitution preserves mobility)**.** If $\mathsf{Mob}\,(A)$ then $\mathsf{Mob}\,([a/x]A)$.

Lemma 5.2.8 ($\Rrightarrow$ preserves valuehood)**.** If $v \Rrightarrow a$ then a is a value.

Lemma 5.2.9 ($\Rrightarrow$ preserves valuehood of types)**.** If $V \Rrightarrow A$ then A is a value.

Lemma 5.2.10 ($\Rrightarrow^*$ preserves head forms)**.** If $\mathsf{hd}\,(A) = hf$ and $A \Rrightarrow^* B$ then $\mathsf{hd}\,(B) = hf$.

[1]The mechanized proof for the LF$^\theta$ system from Chapter 4, for example, involves approximately 400 individual lemmas and theorems. So, even stating every lemma needed would be a monumental task.

Lemma 5.2.11. Suppose $a_1 \Rrightarrow a_2$. Then,

- $[a_1/x]b \Rrightarrow [a_2/x]b$ for any b, and
- $[a_1/x]B \Rrightarrow [a_2/x]B$ for any B, and
- $[a_1/x]k \Rrightarrow [a_2/x]k$ for any k.

Lemma 5.2.12. Suppose $A_1 \Rrightarrow A_2$. Then,

- $[A_1/x]b \Rrightarrow [A_2/x]b$ for any b, and
- $[A_1/x]B \Rrightarrow [A_2/x]B$ for any B, and
- $[A_1/x]k \Rrightarrow [A_2/x]k$ for any k.

Lemma 5.2.13. Suppose $a_1 \Rrightarrow^* a_2$. Then,

- $[a_1/x]b \Rrightarrow^* [a_2/x]b$ for any b, and
- $[a_1/x]B \Rrightarrow^* [a_2/x]B$ for any B, and
- $[a_1/x]k \Rrightarrow^* [a_2/x]k$ for any k.

Lemma 5.2.14. Suppose $A_1 \Rrightarrow^* A_2$. Then,

- $[A_1/x]b \Rrightarrow^* [A_2/x]b$ for any b, and
- $[A_1/x]B \Rrightarrow^* [A_2/x]B$ for any B, and
- $[A_1/x]k \Rrightarrow^* [A_2/x]k$ for any k.

Lemma 5.2.15. For any value v and variable x,

- If $a \Rrightarrow^* b$ then $[v/x]a \Rrightarrow^* [v/x]b$, and
- If $A \Rrightarrow^* B$ then $[v/x]A \Rrightarrow^* [v/x]B$, and
- If $k_1 \Rrightarrow^* k_2$ then $[v/x]k_1 \Rrightarrow^* [v/x]k_2$.

For types, it will be convenient to generalize this last lemma slightly to the case where the type is not a value itself but does reduce to a value. We obtain a similar result, since in any case of the original reduction that made use of the fact that x is a value, we may insert some extra reduction steps to reduce the substituted type to a value.

Lemma 5.2.16. Suppose $B \Rrightarrow^* V$ for some type B and type value V. Then, for any variable x,

- If $a_1 \Rrightarrow^* a_2$ then $[B/x]a_1 \Rrightarrow^* [V/x]a_2$, and

- If $A_1 \Rrightarrow^* A_2$ then $[B/x]A_1 \Rrightarrow^* [V/x]A_2$, and
- If $k_1 \Rrightarrow^* k_2$ then $[B/x]k_1 \Rrightarrow^* [V/x]k_2$.

Lemma 5.2.17 (Confluence)**.** If $A \Rrightarrow^* A_1$ and $A \Rrightarrow^* A_2$, then there is some B such that $A_1 \Rrightarrow^* B$ and $A_2 \Rrightarrow^* B$.

Lemma 5.2.18 ($\rightsquigarrow$ is a subrelation of $\Rrightarrow$)**.**

- If $a \rightsquigarrow b$, then $a \Rrightarrow b$.
- If $A \rightsquigarrow B$, then $A \Rrightarrow B$.

Lemma 5.2.19 ($\rightsquigarrow^*$ and $\Rrightarrow^*$ agree on normalization)**.** If $A \Rrightarrow^* V$ then there is some value V' such that $A \rightsquigarrow^* V'$.

5.2.2 Typing Basics

In this section, we prove several preliminaries about the typing relation that are necessary for the substitution, inversion, and preservation results to come. These lemmas are mostly proved by mutual induction on the three typing judgements, so we state all three parts together. Several of them (weakening, substitution) additionally have two versions: one for terms and one for types.

Lemma 5.2.20 (Term variable weakening)**.** Suppose $\Gamma \vdash B : \star_\sigma$ and $x \notin \mathsf{dom}(\Gamma_1, \Gamma_2)$.

- If $\Gamma_1, \Gamma_2 \vdash^\theta a : A$ then $\Gamma_1, x : B, \Gamma_2 \vdash^\theta a : A$.
- If $\Gamma_1, \Gamma_2 \vdash A : k$ then $\Gamma_1, x : B, \Gamma_2 \vdash A : k$.
- If $\Gamma_1, \Gamma_2 \vdash k$ then $\Gamma_1, x : B, \Gamma_2 \vdash k$.

Proof. By mutual induction on the three typing derivations. □

Lemma 5.2.21 (Type variable weakening)**.** Suppose $\Gamma \vdash k$ and $x \notin \mathsf{dom}(\Gamma_1, \Gamma_2)$.

- If $\Gamma_1, \Gamma_2 \vdash^\theta a : A$ then $\Gamma_1, x : k, \Gamma_2 \vdash^\theta a : A$.
- If $\Gamma_1, \Gamma_2 \vdash A : k$ then $\Gamma_1, x : k, \Gamma_2 \vdash A : k$.
- If $\Gamma_1, \Gamma_2 \vdash k$ then $\Gamma_1, x : k, \Gamma_2 \vdash k$.

Proof. By mutual induction on the three typing derivations. □

Lemma 5.2.22 (Context regularity)**.** For any Γ, θ, a, A, and k,

- If $\Gamma \vdash^\theta a : A$ then $\vdash \Gamma$.
- If $\Gamma \vdash A : k$ then $\vdash \Gamma$.

- If $\Gamma \vdash k$ then $\vdash \Gamma$.

Proof. By mutual induction on the three typing derivations. In the case of TMu, it is also necessary to observe that, if $\vdash \Gamma, x : k$, then $\vdash \Gamma$. This is immediate by inversion of the definition of $\vdash \Gamma$. □

Lemma 5.2.23 (Context inversion for term variables)**.** If $\vdash \Gamma_1, x : A, \Gamma_2$, then $\Gamma_1 \vdash A : \star_\sigma$.

Proof. By induction on the derivation of $\vdash \Gamma_1, x : A, \Gamma_2$. □

Lemma 5.2.24 (Context inversion for type variables)**.** If $\vdash \Gamma_1, x : k, \Gamma_2$, then $\Gamma_1 \vdash k$.

Proof. By induction on the derivation of $\vdash \Gamma_1, x : k, \Gamma_2$. □

Lemma 5.2.25 (Regularity)**.** For any Γ, θ, a, A and k,

- If $\Gamma \vdash^\theta a : A$ then $\Gamma \vdash A : \star_\sigma$.
- If $\Gamma \vdash A : k$ then $\Gamma \vdash k$ or $k = \star_\sigma$.

Proof. By mutual induction on the derivations $\mathcal{D} :: \Gamma \vdash^\theta a : A$ and $\mathcal{E} :: \Gamma \vdash A : k$. In almost all cases, the desired result is available immediately as an induction hypothesis or premise of the typing rule. In a few cases, it is also necessary to employ TMonoPoly. The remaining cases are straightforward.

- $\mathcal{D} = \dfrac{(x : A) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash^\theta x : A}$ EVar

  We must show that $\Gamma \vdash A : \star_\sigma$. We know that Γ has the form $\Gamma_1, x : A, \Gamma_2$ for some Γ_1 and Γ_2. By Lemma 5.2.23, it follows that $\Gamma_1 \vdash A : \star_\sigma$. The desired result then follows by weakening (Lemma 5.2.20).

- $\mathcal{D} = \dfrac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Z} : \mathsf{Nat}}$ EZero

  We must show $\Gamma \vdash \mathsf{Nat} : \star_\sigma$, which is immediate by TNat and TMonoPoly.

- $\mathcal{D} = \dfrac{\begin{array}{c} a \Rrightarrow^* c \qquad b \Rrightarrow^* c \\ \Gamma \vdash^{\theta_1} a : A \qquad \Gamma \vdash^{\theta_2} b : B \end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = b}$ ERefl

  We must show that $\Gamma \vdash a = b : \star_\sigma$. By TMonoPoly, it will be enough to show that $\Gamma \vdash a = b : \star_\tau$. By TEq, it is enough to show that a and b have types in the programmatic fragment. This follows from the two typing hypotheses of ERefl, using ESub when θ_1 or θ_2 is L.

- $\mathcal{D} = \dfrac{\begin{array}{c} A \Rrightarrow^* C \quad B \Rrightarrow^* C \\ \Gamma \vdash A : k_1 \quad \Gamma \vdash B : k_2 \end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{trefl} : A = B}$ EReflT

  We must show that $\Gamma \vdash A = B : \star_\sigma$. By TMonoPoly, it is enough to show that $\Gamma \vdash A = B : \star_\tau$. By TTEq, this follows directly from the two kinding hypotheses of EReflT.

- $\mathcal{D} = \dfrac{\begin{array}{c} \mathcal{D}' \\ \Gamma \vdash^\theta a : A \end{array}}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}$ EBoxP

  We must show that $\Gamma \vdash A@\theta : \star_\sigma$. By TAt, it is is enough to show that $\Gamma \vdash A : \star_\sigma$, which is precisely the induction hypothesis for $\mathcal{D}'$.

- EBoxL and EBoxLV are similar to the previous case.

- $\mathcal{D} = \dfrac{(x : k) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : k}$ TVar

  We must show that $\Gamma \vdash k$. We know that Γ has the form $\Gamma_1, x : k, \Gamma_2$ for some Γ_1 and Γ_2. By Lemma 5.2.24, it follows that $\Gamma_1 \vdash k$. The desired result then follows by weakening (Lemma 5.2.21).

- $\mathcal{D} = \dfrac{\begin{array}{c} \Gamma \vdash A : s \quad \mathsf{Mob}\,(A) \\ \Gamma, x : A \vdash B : s \end{array}}{\Gamma \vdash (x : A) \to B : s}$ TArrComp

  We must show that either $\Gamma \vdash s$ or s is $\star_\sigma$. But s is either $\star_\tau$ or $\star_\sigma$. In the latter case the result is immediate. If s is $\star_\tau$, then by rule KSort it will be enough to show $\vdash \Gamma$, which follows by context regularity (Lemma 5.2.22).

- TSigma, TSum and TAt are similar to the previous case.

□

Lemma 5.2.26 ($\star_\sigma$ is untyped)**.** There is no derivation of $\Gamma \vdash \star_\sigma$.

Proof. By inspection of the inference rules for $\Gamma \vdash k$. □

Lemma 5.2.27 (Context conversion for terms)**.** Suppose $\Gamma_1 \vdash^{\mathsf{L}} p : b_1 = b_2$ and $\Gamma_1 \vdash [b_2/y]B : \star_\sigma$.

- If $\Gamma_1, x : [b_1/y]B, \Gamma_2 \vdash^\theta a : A$ then $\Gamma_1, x : [b_2/y]B, \Gamma_2 \vdash^\theta a : A$.

- If $\Gamma_1, x : [b_1/y]B, \Gamma_2 \vdash A : k$ then $\Gamma_1, x : [b_2/y]B, \Gamma_2 \vdash A : k$.

- $\Gamma_1, x : [b_1/y]B, \Gamma_2 \vdash k$ then $\Gamma_1, x : [b_2/y]B, \Gamma_2 \vdash k$.

Proof. By mutual induction on the three typing derivations. The only interesting case is the one for term variables:

- $\mathcal{D} = \dfrac{(z : A) \in \Gamma_1, x : [b_1/y]B, \Gamma_2 \qquad \vdash \Gamma_1, x : [b_1/y]B, \Gamma_2}{\Gamma_1, x : [b_1/y]B, \Gamma_2 \vdash^{\theta} z : A}\ \textsc{EVar}$

  Either $z = x$ or not. If not, the result immediate. So suppose $z = x$. Then A is $[b_1/y]B$ and we must show that $\Gamma_1, x : [b_2/y]B, \Gamma_2 \vdash^{\theta} z : [b_1/y]B$. But $\Gamma_1, x : [b_2/y]B, \Gamma_2 \vdash^{\theta} z : [b_2/y]B$ by EVar, and the result follows by EConv.

□

Lemma 5.2.28 (Value boxing)**.** If $\Gamma \vdash^{\theta'} v : A$ then $\Gamma \vdash^{\theta} v : A@\theta'$, for any θ and θ'.

Proof. We break down the possible logicalities individually:

- If $\theta = \mathsf{P}$, this is an instance of EBoxP.
- If $\theta = \theta' = \mathsf{L}$, this is an instance of EBoxL.
- If $\theta = \mathsf{L}$ and $\theta' = \mathsf{P}$, this is an instance of EBoxLV.

□

Lemma 5.2.29. If $\vdash \Gamma_1, x : A, \Gamma_2$ then $\mathsf{Mob}\,(A)$.

Proof. By induction on the derivation of $\vdash \Gamma_1, x : A, \Gamma_2$. □

5.2.3 Substitution

The substitution lemmas for PCC$^{\theta}$ are unsurprising. There are two lemmas—one for substituting terms and one for substituting types.

Lemma 5.2.30 (Term Substitution)**.** Suppose $\Gamma_1 \vdash^{\theta'} v : B$.

- If $\Gamma_1, x : B, \Gamma_2 \vdash^{\theta} a : A$, then $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta} [v/x]a : [v/x]A$.
- If $\Gamma_1, x : B, \Gamma_2 \vdash A : k$, then $\Gamma_1, [v/x]\Gamma_2 \vdash [v/x]A : [v/x]k$.
- If $\Gamma_1, x : B, \Gamma_2 \vdash k$, then $\Gamma_1, [v/x]\Gamma_2 \vdash [v/x]k$.

Proof. By mutual induction on the three derivations $\mathcal{D} :: \Gamma_1, x : B, \Gamma_2 \vdash^{\theta} a : A$, $\mathcal{E} :: \Gamma_1, x : B, \Gamma_2 \vdash A : k$ and $\mathcal{F} :: \Gamma_1, x : B, \Gamma_2 \vdash k$. Most cases are immediate by induction. In typing rules with a value restriction, the additional observation that substituting a value into a value yields a value is necessary (Lemma 5.2.6). A few cases are (slightly) more interesting:

- $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}' \\ \Gamma_1, x:B, \Gamma_2 \vdash^{\mathsf{L}} a_1 : B_1 = B_2 \qquad \Gamma_1, x:B, \Gamma_2 \vdash^{\theta_1} a : A \\ \mathsf{hd}\,(B_1) = \mathit{hf}_1 \qquad \mathsf{hd}\,(B_2) = \mathit{hf}_2 \qquad \mathit{hf}_1 \neq \mathit{hf}_2 \\ \Gamma_1, x:B, \Gamma_2 \vdash A : \star_\sigma \qquad \Gamma_1, x:B, \Gamma_2 \vdash A' : \star_\sigma\end{array}}{\Gamma_1, x:B, \Gamma_2 \vdash^{\theta_2} a : A'}\ \textsc{EContra}$$

  We must show that $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta_2} [v/x]a : [v/x]A'$. This is almost immediate by an application of EContra, using the induction hypotheses for the premises above. However, observe that the induction hypothesis for $\mathcal{D}'$ yields $\Gamma_1, [v/x]\Gamma_2 \vdash^{\mathsf{L}} [v/x]a_1 : ([v/x]B_1) = ([v/x]B_2)$, so we must show that $[v/x]B_1$ and $[v/x]B_2$ have different head forms. But by Lemma 5.2.5, $\mathsf{hd}\,([v/x]B_1) = \mathit{hf}_1$, and $\mathsf{hd}\,([v/x]B_2) = \mathit{hf}_2$. The required result follows from the hypothesis that $\mathit{hf}_1 \neq \mathit{hf}_2$.

- $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{E}' \\ \Gamma_1, x:B, \Gamma_2 \vdash [v'/y]A_2 = [v'/y]B_2 : \star_\tau \\ \Gamma_1, x:B, \Gamma_2 \vdash^{\theta} a : (\Sigma y{:}A_1 \,.\, A_2) = (\Sigma x{:}B_1 \,.\, B_2) \\ \Gamma_1, x:B, \Gamma_2 \vdash^{\theta'} v' : A_1\end{array}}{\Gamma_1, x:B, \Gamma_2 \vdash^{\theta} a : [v'/y]A_2 = [v'/y]B_2}\ \textsc{ESigmaInv2}$$

  We must show that $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta} [v/x]a : [v/x][v'/y]A_2 = [v/x][v'/y]B_2$. Intuitively, the induction hypotheses and ESigmaInv2 itself should provide this result, but neither our goal nor the IH for $\mathcal{E}'$ have quite the right syntactic form for that inference rule—the substitution for y is in the wrong place.

  To make progress, observe that $[v/x][v'/y]A_2 = [[v/x]v'/y][v/x]A_2$, since we me assume y to be fresh for v. A similar equality holds for B_2. Additionally, $[v/x]v'$ is a value by Lemma 5.2.6. Having observed this, the IH for $\mathcal{E}'$ may be rewritten as:

  $$\Gamma_1, [v/x]\Gamma_2 \vdash [[v/x]v'/y][v/x]A_2 = [[v/x]v'/y][v/x]B_2 : \star_\tau$$

  The desired result then follows by ESigmaInv2 and the other induction hypotheses.

- Cases EArrInv2, EMuInv, EAppPoly, EAppComp, TAppTLC, TAppDep, EPair, ERoll, EUnroll, EConv, EConvT, TConv, and TConvT require similar substitution juggling but are otherwise straightforward.

- Cases ERefl and EReflT require Lemma 5.2.15 but are otherwise straightforward.

- $$\mathcal{D} = \cfrac{(y:A) \in \Gamma_1, x:B, \Gamma_2 \qquad \vdash \Gamma_1, x:B, \Gamma_2}{\Gamma_1, x:B, \Gamma_2 \vdash^{\theta} y : A}\ \textsc{EVar}$$

  Either $x = y$ or not.

  - Suppose $x = y$. Then $A = B$, since the $\vdash \Gamma$ judgement ensures no variables are repeated in a context. We must show that $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta} v : [v/x]B$. We know by assumption that $\Gamma_1 \vdash^{\theta'} v : B$. Since $\vdash \Gamma_1, x : B, \Gamma_2$ and variables are not repeated in well-formed contexts, $[v/x]B = B$. By weakening, we therefore have $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta'} v : [v/x]B$.

    If $\theta = \theta'$, we are done. If θ is P and θ' is L, the result follows by TSub. If θ is L and P, the result follows by EMVal, since Lemmas 5.2.29 and 5.2.7 yield $\mathsf{Mob}\,(B)$.

  - Suppose $x \neq y$. We must show that $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta} y : [v/x]A$. We know the binding $y : A$ occurs in Γ_1 or Γ_2. If it occurs in Γ_2, rule EVar yields the desired result immediately. If it occurs in Γ_1 then x does not occur free in A, since variables are not repeated in contexts. Thus $[v/x]A = A$, and the result follows by EVar.

- TVar is similar (but simpler, since it cannot be the case that $x = y$).

□

Lemma 5.2.31 (Type Substitution)**.** Suppose $\Gamma_1 \vdash V : k'$.

- If $\Gamma_1, x : k', \Gamma_2 \vdash^{\theta} a : A$, then $\Gamma_1, [V/x]\Gamma_2 \vdash^{\theta} [V/x]a : [V/x]A$.
- If $\Gamma_1, x : k', \Gamma_2 \vdash A : k$, then $\Gamma_1, [V/x]\Gamma_2 \vdash [V/x]A : [V/x]k$.
- If $\Gamma_1, x : k', \Gamma_2 \vdash k$, then $\Gamma_1, [V/x]\Gamma_2 \vdash [V/x]k$.

Proof. The proof is similar to the previous lemma. □

Interestingly, it is also the case that non-value types may be substituted, as long as they do reduce to a value. Intuitively, this is because the only value restrictions on types occur in the reduction relations, and the typing relations do not depend on the particular number steps a reduction takes.

Lemma 5.2.32 (Type substitution)**.** Suppose $\Gamma_1 \vdash B : k'$ and $B \Rrightarrow^* V$.

- If $\Gamma_1, x : k', \Gamma_2 \vdash^{\theta} a : A$, then $\Gamma_1, [B/x]\Gamma_2 \vdash^{\theta} [B/x]a : [B/x]A$.
- If $\Gamma_1, x : k', \Gamma_2 \vdash A : k$, then $\Gamma_1, [B/x]\Gamma_2 \vdash [B/x]A : [B/x]k$.
- If $\Gamma_1, x : k', \Gamma_2 \vdash k$, then $\Gamma_1, [B/x]\Gamma_2 \vdash [B/x]k$.

Proof. Most cases are identical to the type substitution result (Lemma 5.2.31), since there are no value restrictions on types, except in the operational semantics. In the two cases that refer to the operational semantics (TRefl and TReflT), Lemma 5.2.16 justifies the substitution of a non-value type. □

5.2.4 Inversion and Preservation

In this section, we prove the inversion and preservation lemmas for PCC$^\theta$. While the statement of some of these lemmas is somewhat unusual, all of the novel aspects have been motivated and explained in Chapter 4. So, we focus here on detailed proofs rather than exposition of the technique. We also include several facts about PCC$^\theta$'s notion of equality which could not be proved until after inversion for equality types.

Lemma 5.2.33 (Inversion for term equality)**.** If $\Gamma \vdash a = b : k$, then $\Gamma \vdash^{\theta_1} a : A$ and $\Gamma \vdash^{\theta_2} b : B$ for some θ_1, θ_2, A and B.

Proof. By induction on the derivation of $\Gamma \vdash a = b : k$. □

Lemma 5.2.34 (Inversion for type equality)**.** If $\Gamma \vdash A = B : k$, then $\Gamma \vdash A : k_1$ and $\Gamma \vdash B : k_2$ for some k_1 and k_2.

Proof. By induction on the derivation of $\Gamma \vdash A = B : k$. □

Lemma 5.2.35 (Type equality is reflexive)**.** If $\Gamma \vdash A : k$, then $\Gamma \vdash^{\mathsf{L}} \mathsf{trefl} : A = A$.

Proof. Immediate by EReflT, since $A \Rrightarrow^* A$. □

Lemma 5.2.36 (Type equality is symmetric)**.** If $\Gamma \vdash^{\mathsf{L}} p : A = B$ then $\Gamma \vdash^{\mathsf{L}} p : B = A$.

Proof. By regularity (Lemma 5.2.25) and inversion for type equality (Lemma 5.2.34), we have $\Gamma \vdash A : k_1$ and $\Gamma \vdash B : k_2$ for some k_1, and k_2. It follows that $\Gamma \vdash^{\mathsf{L}} \mathsf{trefl} : A = A$, because equality is reflexive (Lemma 5.2.35). The result then follows by TConvT, since $A = A$ may be written $[A/x](x = A)$. □

Lemma 5.2.37 (Term equality is reflexive)**.** If $\Gamma \vdash^{\theta} a : A$, then $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = a$.

Proof. Immediate by ERefl, since $a \Rrightarrow^* a$. □

Lemma 5.2.38 (Term equality is symmetric)**.** If $\Gamma \vdash^{\mathsf{L}} p : a = b$ then $\Gamma \vdash^{\mathsf{L}} p : b = a$.

Proof. By regularity (Lemma 5.2.25) and inversion for term equality (Lemma 5.2.33), we have $\Gamma \vdash^{\theta_1} a : A$ and $\Gamma \vdash^{\theta_2} b : B$ for some θ_1, θ_2, A, and B. It follows that $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = a$, because equality is reflexive (Lemma 5.2.37). The result then follows by TConv, since $a = a$ may be written $[a/x](x = a)$. □

Lemma 5.2.39 (@ equality construction)**.** Suppose $\Gamma \vdash^{\mathsf{L}} p : A = B$. If $\Gamma \vdash A : s_1$ and $\Gamma \vdash B : s_2$ then $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : (A@\theta) = (B@\theta)$.

Proof. By TAt, $\Gamma \vdash A@\theta : s_1$. Since equality is reflexive (Lemma 5.2.35), $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : A@\theta = A@\theta$. The result then follows by TConvT. □

Lemma 5.2.40 (Inversion for @-types)**.** If $\Gamma \vdash A@\theta : k$, then k is a sort s and $\Gamma \vdash A : s$.

Proof. By induction on the derivation $\mathcal{E} :: \Gamma \vdash A@\theta : k$. Most cases are ruled out because the subject does not have the correct form. Of the remaining cases, the premises of TAT satisfy the lemma directly, and TMONOPOLY is immediate by induction (and another use of TMONOPOLY). The two conversion cases remain:

- $$\mathcal{E} = \dfrac{\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \qquad \overset{\mathcal{E}'}{\Gamma \vdash A@\theta : [b_1/x]k} \qquad \Gamma \vdash [b_2/x]k}{\Gamma \vdash A@\theta : [b_2/x]k}\ \textsc{TConv}$$

  Here, the IH for $\mathcal{E}'$ yields that $[b_1/x]k$ is s for some sort s, and $\Gamma \vdash A : s$. But if $[b_1/x]k = s$, then it must be the case that $k = s$, since b_1 cannot be a sort. It follows that $[b_2/x]k$ is s, as desired.

- Case TCONVT is similar.

□

Lemma 5.2.41 (Inversion for computational arrow types)**.** If $\Gamma \vdash (x : A) \to B : k$, then $\mathsf{Mob}\,(A)$ and k is a sort s such that $\Gamma \vdash A : s$ and $\Gamma, x : A \vdash B : s$.

Proof. Similar to inversion for @-types (Lemma 5.2.40). □

Lemma 5.2.42 (Inversion for Σ-types)**.** If $\Gamma \vdash \Sigma x : A \,.\, B : k$ then $\mathsf{Mob}\,(A)$ and k is a sort s such that $\Gamma \vdash A : s$ and $\Gamma, x : A \vdash B : s$.

Proof. Similar to inversion for @-types (Lemma 5.2.40). □

Lemma 5.2.43 (Inversion for dependent arrow kinds)**.** If $\Gamma \vdash (x : A) \to k$ then $\Gamma \vdash A : \star_\sigma$ and $\Gamma, x : A \vdash k$.

Proof. By inspection of the $\Gamma \vdash k$ judgement. □

Lemma 5.2.44 (Inversion for type-level computation arrow kinds)**.** If $\Gamma \vdash (x : k_1) \to k_2$ then $\Gamma \vdash k_1$ and $\Gamma, x : k_1 \vdash k_2$.

Proof. By inspection of the $\Gamma \vdash k$ judgement. □

Because PCC$^\theta$ includes no notion of kind equality, it is most convenient to state inversion for type-level functions directly in the form that will be needed for preservation (i.e., to explicitly mention the substitution of a value into the body of the function).

Lemma 5.2.45 (Inversion for type-level functions)**.** Suppose $\Gamma \vdash \lambda x : k.B : (x : k_1) \to k_2$. Then, for any V, if $\Gamma \vdash V : k_1$, then $\Gamma \vdash [V/x]B : [V/x]k_2$

Proof. By induction on the derivation $\mathcal{E} :: \Gamma \vdash \lambda x : k.B : (x : k_1) \to k_2$. Most cases do not apply because the conclusion does not have the correct form. Case TLamTLC is immediate by the substitution lemma for types (Lemma 5.2.31). There are only two interesting cases:

- $$\mathcal{E} = \dfrac{\Gamma \vdash^{\mathsf{L}} p : b_1 = b_2 \qquad \overset{\mathcal{E}'}{\Gamma \vdash \lambda x : k.B : [b_1/y]k'} \qquad \Gamma \vdash [b_2/y]k'}{\Gamma \vdash \lambda x : k.B : [b_2/y]k'}\ \textsc{TConv}$$

  Observe first that the conclusion of this rule assigns the kind $[b_2/y]k'$, but by assumption this is equal to $(x : k_1) \to k_2$, so it must be the case that $k' = (x : k_1') \to k_2'$ for some kinds k_1' and k_2', since b_2 is not a kind.

  Let some V be given such that $\Gamma \vdash V : [b_2/y]k_1'$. We must show that $\Gamma \vdash [V/x]B : [V/x][b_2/y]k_2'$. Since equality is symmetric (Lemma 5.2.38), we may use TConv to observe that $\Gamma \vdash V : [b_1/y]k_1'$. It then follows by the IH for $\mathcal{E}'$ that $\Gamma \vdash [V/x]B : [V/x][b_1/y]k_2'$.

  Since we may pick x and y such that they do not appear free in V or b_1, it is the case that $[V/x][b_1/y]k_2' = [b_1/y][V/x]k_2'$. Thus, by another use of TConv, we obtain $\Gamma \vdash [V/x]B : [b_2/y][V/x]k_2'$. And since $[b_2/y][V/x]k_2' = [V/x][b_2/y]k_2'$, this concludes the case.

- Case TConvT is similar.

□

Lemma 5.2.46 (Inversion for term-level λs)**.** Suppose $\Gamma \vdash^{\theta} \lambda x : B_1.b : B$. Then either:

(1) $\Gamma \vdash^{\mathsf{L}} p : B = (x : B_1) \to B_2$ for some p and B_2 such that $\Gamma, x : B_1 \vdash^{\theta} b : B_2$,

(2) or, $\Gamma \vdash^{\mathsf{L}} p : B = (...(((x : B_1) \to B_2)@\theta_1)...)@\theta_n$ for some p, B_2, and θ_1,...θ_n such that $\Gamma, x : B_1 \vdash^{\theta_1} b : B_2$.

Proof. By induction on on the derivation $\mathcal{D} :: \Gamma \vdash^{\theta} \lambda x{:}B_1.b : B$. Many cases are ruled out because the subject of the derivation does not have the right form, and several others are immediate by induction. We consider the remaining cases individually:

- $$\mathcal{D} = \dfrac{\begin{array}{c}\Gamma \vdash^{\mathsf{L}} a_1 : A_1 = A_2 \qquad \overset{\mathcal{D}'}{\Gamma \vdash^{\theta'} \lambda x{:}B_1.b : A} \\ \mathsf{hd}\,(A_1) = hf_1 \qquad \mathsf{hd}\,(A_2) = hf_2 \qquad hf_1 \neq hf_2 \\ \Gamma \vdash A : \star_\sigma \qquad \Gamma \vdash B : \star_\sigma\end{array}}{\Gamma \vdash^{\theta} \lambda x{:}B_1.b : B}\ \textsc{EContra}$$

  In this case, we will use EContra to construct the necessary derivations to satisfy condition (1). We begin by observing that the IH for $\mathcal{D}'$ yields, in either case, two derivations: one has the form $\Gamma, x : B_1 \vdash^{\theta'} b : B_2'$ for some θ' and B_2', and the other is the proof of an equality which, via regularity and inversion for equality types and @-types (Lemmas 5.2.25, 5.2.34, and 5.2.40), yields a proof of $\Gamma \vdash (x : B_1) \to B_2' : k$ for some kind k.

In satisfying condition (1), we pick Z for p and B_2' for B_2. We must prove two things:

– $\Gamma \vdash^{\mathsf{L}} \mathsf{Z} : B = (x : B_1) \to B_2'$.

  This will follow from a use of ECONTRA using the same contradictory equality. The three remaining hypotheses of ECONTRA are that Z is a well-typed term in context Γ, that its type has kind $\star_\sigma$, and that $B = (x : B_1) \to B_2'$ has kind $\star_\sigma$. The first two requirements are simple by EZERO, TNAT, TMONOPOLY, and regularity (Lemma 5.2.25).

  To show $\Gamma \vdash B = (x : B_1) \to B_2' : \star_\sigma$, observe that, by rules TEQT and TMONOPOLY, it is enough to show that B and $(x : B_1) \to B_2'$ are well-kinded in context Γ. The former is a premise of the use of ECONTRA above, and we already observed that the latter is a consequence of the IH for $\mathcal{D}'$ in either case.

– $\Gamma, x : B_1 \vdash^{\theta} b : B_2'$.

  We already observed that, for some θ', $\Gamma, x : B_1 \vdash^{\theta'} b : B_2'$ is a consequence of the IH in either case. Since we know there is a contradiction in the system, rule ECONTRA may be used directly to "switch fragments" to θ, as long as $\Gamma, x : B_1 \vdash B_2 : \star_\sigma$. But we already observed that $\Gamma \vdash (x : B_1) \to B_2' : k$, so this follows from inversion for arrow types (Lemma 5.2.41) and possibly a use of TMONOPOLY.

• $$\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Gamma \vdash^{\theta} \lambda x{:}B_1.b : (A_1@\theta') = (A_2@\theta')} \qquad \Gamma \vdash A_1 = A_1 : \star_\tau}{\Gamma \vdash^{\theta} \lambda x{:}B_1.b : A_1 = A_2}\ \text{EATINV}$$

In this case, the IH for $\mathcal{D}'$ yields either a proof that an equality type is equal to an arrow type or a proof that an equality type is equal to an @-type. In either case this is an internal contradiction, since equality, arrow and @-types all have different head forms. With a contradiction in hand, this case proceeds similarly to the previous case, ECONTRA.

• Cases EARRINV1, EARRINV2, ETARRINV2, ESUMINV1, ESUMINV2, ESIGMAINV1, ESIGMAINV2 and EMUINV are similar to EATINV.

• $$\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Gamma \vdash^{\theta'} \lambda x{:}B_1.b : B@\theta} \qquad \Gamma \vdash B : \star_\sigma}{\Gamma \vdash^{\theta} \lambda x{:}B_1.b : B}\ \text{EUNBOXVAL}$$

The IH for $\mathcal{D}'$ yields two cases. In case (1), we have $B@\theta'$ is provably equal to an arrow type. This is a contradiction in the system, and we may reason as in the case above for EContra.

Otherwise we have $\Gamma \vdash^{\mathsf{L}} p : B@\theta = (...(((x : B_1) \to B_2)@\theta_1)...)@\theta_n$ for some p, B_2, and θ_1,...θ_n such that $\Gamma, x : B_1 \vdash^{\theta_1} b : B_2$. Either $\theta = \theta_n$ or not. If not, we have a contradiction in the system and may conclude as in the case for EContra above.

If it is the case that $\theta = \theta_n$, then by rule EAtInv we obtain $\Gamma \vdash^{\mathsf{L}} p : B = (...(((x : B_1) \to B_2)@\theta_1)...)@\theta_{n-1}$. Now, if $n - 1 > 0$, this satisfies condition (2). Otherwise, n was 1, in which we actually have $\Gamma \vdash^{\mathsf{L}} p : B = (x : B_1) \to B_2$, and we have satisfied condition (1) (since $\theta_1 = \theta$).

- $\mathcal{D} = \dfrac{\Gamma, x : B_1 \vdash^{\mathsf{L}} b : B_2 \qquad \Gamma \vdash (x : B_1) \to B_2 : \star_\sigma}{\Gamma \vdash^{\mathsf{L}} \lambda x{:}B_1.b : (x : B_1) \to B_2}$ ELamComp

  Condition (1) is immediately satisfied, observing that type equality is reflexive (Lemma 5.2.35).

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Gamma \vdash^{\mathsf{P}} v : B} \qquad \mathsf{Mob}\,(B)}{\Gamma \vdash^{\mathsf{L}} v : B}$ EMVal

  The IH for $\mathcal{D}'$ yields two cases. In the first case, we know that B is provably equal to an arrow type. But if we consider the possible derivations of $\mathsf{Mob}\,(B)$, we find B must have a head form that is not an arrow. Thus, there is a contradiction in the system and we may proceed as in the case for EContra above.

  In the second case, the result is immediate since condition (2) does not mention the logicality θ from the typing derivation.

- $\mathcal{D} = \dfrac{\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \qquad \overset{\mathcal{D}'}{\Gamma \vdash^{\theta} \lambda x{:}B_1.b : [b_1/y]B} \qquad \Gamma \vdash [b_2/y]B : \star_\sigma}{\Gamma \vdash^{\theta} \lambda x{:}B_1.b : [b_2/y]B}$ EConv

  We consider the two cases of the IH for $\mathcal{D}'$ individually:

  (1) Suppose the IH yields that $\Gamma \vdash^{\mathsf{L}} p : [b_1/y]B = (x : B_1) \to B_2$ for some p and B_2 such that $\Gamma, x : B_1 \vdash^{\theta} b : B_2$. Then to satisfy condition (1), it will be enough to show that $\Gamma \vdash^{\mathsf{L}} p : [b_2/y]B = (x : B_1) \to B_2$. But we may pick y to be free for $(x : B_1) \to B_2$ so that $[b_2/y]B = (x : B_1) \to B_2$ is the same as $[b_2/y](B = (x : B_1) \to B_2)$. So the desired equality follows directly from a use of EConv on the previous proof that $\Gamma \vdash^{\mathsf{L}} p : [b_1/y]B = (x : B_1) \to B_2$.

  (2) Similar to (1).

- Case EConvT is similar to EConv.

- $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}' \\ \Gamma \vdash^{\mathsf{L}} \lambda x\!:\!B_1.b : B\end{array}}{\Gamma \vdash^{\mathsf{L}} \lambda x\!:\!B_1.b : B@\theta}\ \text{EBoxL}$$

  We consider the two possibilities for $\mathcal{D}'$'s IH individually.

  (1) Suppose the IH yields that $\Gamma \vdash^{\mathsf{L}} p : B = (x : B_1) \to B_2$ for some p and B_2 such that $\Gamma, x : B_1 \vdash^{\mathsf{L}} b : B_2$. By Lemma 5.2.39, $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : B@\theta = ((x : B_1) \to B_2)@\theta$, so condition (2) is satisfied if we can show that $\Gamma, x : B_1 \vdash^{\theta} b : B_2$, which we already know if θ is L, and follows by TSub otherwise.

  (2) Similar to (1).

- Cases EBoxP and EBoxLV are similar to EBoxL.

□

Lemma 5.2.47 (Inversion for λ-expressions at arrow types)**.** Suppose $\Gamma \vdash^{\theta} \lambda x\!:\!B_1'.b : (x : B_1) \to B_2$ and $\Gamma \vdash^{\theta} v : B_1$. Then $\Gamma \vdash^{\theta} [v/x]b : [v/x]B_2$.

Proof. According to the inversion lemma for λ expressions (Lemma 5.2.46), there are two possibilities. We consider them individually.

(1) Suppose the lemma yields that $\Gamma \vdash^{\mathsf{L}} p : (x : B_1) \to B_2 = (x : B_1') \to B_2'$ for some p and B_2' such that $\Gamma, x : B_1' \vdash^{\theta} b : B_2'$. By rule EArrInv1, we have $\Gamma \vdash^{\mathsf{L}} p : B_1 = B_1'$, so by EConvT we know $\Gamma \vdash^{\theta} v : B_1'$. Then substitution (Lemma 5.2.30) yields $\Gamma \vdash^{\theta} [v/x]b : [v/x]B_2'$. Rule EArrInv2 provides a proof that $\Gamma \vdash^{\mathsf{L}} p : [v/x]B_2 = [v/x]B_2'$, allowing us to conclude the case by rule TConvT.

(2) Suppose instead that the inversion lemma yields that $\Gamma \vdash^{\mathsf{L}} p : (x : B_1) \to B_2 = (...(((x : B_1') \to B_2')@\theta_1)...)@\theta_n$ for some p and B_2' such that $\Gamma, x : B_1' \vdash^{\theta_1} b : B_2'$. In this case we have an equality between two types with different head forms—a contradiction in the system. We may therefore use EContra to obtain $\Gamma \vdash^{\theta} v : B_1'$, and by substitution (Lemma 5.2.30), we find that $\Gamma \vdash^{\theta_1} [v/x]b : [v/x]B_2'$. We conclude by using this and EContra again to derive $\Gamma \vdash^{\theta} [v/x]b : [v/x]B_2$ as desired.

□

Lemma 5.2.48 (Inversion for recursive functions)**.** Suppose $\Gamma \vdash^{\theta} \mathsf{rec}\, f\ (x\!:\!B_1).b : B$. Then either

(1) $\Gamma \vdash^{\mathsf{L}} p : B = (x : B_1) \to B_2$ for some p and B_2 such that $\Gamma, f : (x : B_1) \to B_2, x : B_1 \vdash^{\theta} b : B_2$,

(2) or $\Gamma \vdash^{\mathsf{L}} p : B = (...(((x : B_1) \to B_2)@\theta_1)...)@\theta_n$ for some p, B_2, and θ_1,...θ_n such that $\Gamma, f : (x : B_1) \to B_2, x : B_1 \vdash^{\theta_1} b : B_2$.

Proof. Similar to inversion for lambdas (Lemma 5.2.46). □

Lemma 5.2.49. Suppose $\Gamma \vdash^{\theta} \mathsf{rec}\, f\ (x\!:\!B_1').b : (x : B_1) \to B_2$ and $\Gamma \vdash^{\theta} v : B_1$. Then $\Gamma \vdash^{\theta} [v/x][\mathsf{rec}\, f\ (x\!:\!B_1').b/f]b : [v/x]B_2$.

Proof. Similar to inversion for lambdas at arrow types (Lemma 5.2.47). □

Lemma 5.2.50 (Inversion for pairs)**.** Suppose $\Gamma \vdash^{\theta} \langle a_1, a_2\rangle : B$. Then either:

(1) $\Gamma \vdash^{\mathsf{L}} p : B = \Sigma x : A_1 . A_2$ for some p, A_1 and A_2 such that $\Gamma \vdash^{\theta} a_1 : A_1$ and $\Gamma \vdash^{\theta} a_2 : [a_1/x]A_2$,

(2) or, $\Gamma \vdash^{\mathsf{L}} p : B = (...((\Sigma x : A_1 . A_2)@\theta_1)...)@\theta_n$ for some p, A_1, A_2, and θ_1,...θ_n such that $\Gamma \vdash^{\theta_1} a_1 : A_1$ and $\Gamma \vdash^{\theta_1} a_2 : [a_1/x]A_2$.

Proof. Similar to inversion for lambdas (Lemma 5.2.46). □

Lemma 5.2.51 (Inversion for pairs at Σ-types)**.** If $\Gamma \vdash^{\theta} \langle v_1, a_2\rangle : \Sigma x : A_1 . A_2$, then $\Gamma \vdash^{\theta} v_1 : A_1$ and $\Gamma \vdash^{\theta} a_2 : [v_1/x]A_2$.

Proof. Similar to inversion for lambdas at arrow types (Lemma 5.2.47). The requirement that the first element of the pair is a value is necessary because we must use substitution to show that $[v_1/x]A_2$ is a well-formed type. □

Theorem 5.2.52 (Preservation for parallel reduction)**.**

- If $\Gamma \vdash^{\theta} a : A$ and $a \Rrightarrow a'$ then $\Gamma \vdash^{\theta} a' : A$.
- If $\Gamma \vdash A : k$ and $A \Rrightarrow A'$ then $\Gamma \vdash A' : k$.
- If $\Gamma \vdash k$ and $k \Rrightarrow k'$ then $\Gamma \vdash k'$.

Proof. By mutual induction on the three typing derivations $\mathcal{D} :: \Gamma \vdash^{\theta} a : A$, $\mathcal{E} :: \Gamma \vdash A : k$ and $\mathcal{F} :: \Gamma \vdash k$. Many cases are an immediate consequence of the induction hypotheses. We consider the more interesting cases explicitly. In each case, we examine the possible ways for the subject of the derivation to step via parallel reduction (ignoring PRefl, since the proof is trivial in that case).

- $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\theta} b : (x : A) \to B} \qquad \overset{\mathcal{D}_2}{\Gamma \vdash^{\theta} a : A} \qquad \overset{\mathcal{E}}{\Gamma \vdash [a/x]B : \star_{\sigma}}}{\Gamma \vdash^{\theta} b\ a : [a/x]B}\ \textsc{EAppComp}$$

  There are three possible ways for $b\ a$ to take a step in the parallel reduction relation. We consider them individually.

- Suppose $b\ a \Rrightarrow b'\ a'$ by PApp1, so $b \Rrightarrow b'$ and $a \Rrightarrow a'$. We must show that $\Gamma \vdash^{\theta} b'\ a' : [a/x]B$.

  By Lemma 5.2.11, $[a/x]B \Rrightarrow [a'/x]B$. Thus, the IH for $\mathcal{E}$ yields that $\Gamma \vdash [a'/x]B : \star_{\sigma}$. Putting this together with EAppComp and the IHs for $\mathcal{D}_1$ and $\mathcal{D}_2$, we obtain $\Gamma \vdash^{\theta} b'\ a' : [a'/x]B$.

  By ERefl, $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a' = a$, since $a \Rrightarrow^* a'$ and $a' \Rrightarrow^* a'$. Thus, by EConv, $\Gamma \vdash^{\theta} b'\ a' : [a/x]B$, as desired.

- Suppose b has the form $\lambda x{:}A'.b_1$, a is a value, and $(\lambda x{:}A'.b_1)\ a \Rrightarrow [a'/x]b'$ by PBeta, so $b_1 \Rrightarrow b'$ and $a \Rrightarrow a'$. We must show that $\Gamma \vdash^{\theta} [a'/x]b' : [a/x]B$.

  Now, by Lemma 5.2.8, a' is a value, and by the IH for $\mathcal{D}_2$ we have $\Gamma \vdash^{\theta} a' : A$. Similarly, since $\lambda x{:}A'.b_1 \Rrightarrow \lambda x{:}A'.b'$ by PLam1, the IH for $\mathcal{D}_1$ yields that $\Gamma \vdash^{\theta} \lambda x : A'.b' : (x : A) \to B$. Thus, by inversion for λ-expressions at arrow types (Lemma 5.2.47), we have $\Gamma \vdash^{\theta} [a'/x]b' : [a'/x]B$.

  Since a and a' are both well typed, $a \Rrightarrow^* a'$ and $a' \Rrightarrow^* a'$, TRefl yields $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a' = a$. Thus, by TConv, we have $\Gamma \vdash^{\theta} [a'/x]b' : [a/x]B$ as desired.

- Suppose b has the form $\mathsf{rec}\, f\, (x{:}A').b_1$, a is a value, and $(\mathsf{rec}\, f\, (x{:}A').b_1)\ a \Rrightarrow [a'/x][\mathsf{rec}\, f\, (x{:}A').b'/f]b'$ by PFBeta, so $b_1 \Rrightarrow b'$ and $a \Rrightarrow a'$.

  Now, by Lemma 5.2.8, a' is a value, and by the IH for $\mathcal{D}_2$ we have $\Gamma \vdash^{\theta} a' : A$. Similarly, since $\mathsf{rec}\ f\, (x : A').b_1 \Rrightarrow \mathsf{rec}\ f\, (x : A').b'$ by PFun1, the IH for $\mathcal{D}_1$ yields that $\Gamma \vdash^{\theta} \mathsf{rec}\ f\, (x : A').b' : (x : A) \to B$. Thus, by inversion for recursive functions at arrow types (Lemma 5.2.49), we have $\Gamma \vdash^{\theta} [a'/x][\mathsf{rec}\, f\, (x{:}A').b'/f]b' : [a'/x]B$.

  Since a and a' are both well typed, $a \Rrightarrow^* a'$ and $a' \Rrightarrow^* a'$, TRefl yields $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a' = a$. Thus, by TConv, we have $\Gamma \vdash^{\theta} [a'/x][\mathsf{rec}\, f\, (x{:}A').b'/f]b' : [a/x]B$ as desired.

- Case EAppPoly is similar to EAppComp.

- $$\mathcal{D} = \cfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \vdash^{\theta} a : (\Sigma x{:}A_1\,.\,A_2)@\theta' & \Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a \vdash^{\theta} b : B \end{array} \\ \Gamma \vdash B : \star_{\sigma}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b : B}\ \textsc{PCase}$$

  There are two possible ways for $\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b$ to take a step in the $\Rrightarrow$ relation. We consider them individually.

  - Suppose $(\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b) \Rrightarrow (\mathsf{pcase}_z\ a'\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b')$ by PPCase1, so $a \Rrightarrow a'$ and $b \Rrightarrow b'$.

    By the IHs for $\mathcal{D}_1$ and $\mathcal{D}_2$, we have

$$\Gamma \vdash^{\theta} a' : (\Sigma x{:}A_1\,.\,A_2)@\theta'$$

and:
$$\Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a \vdash^{\theta} b' : B$$

This is almost enough to apply PCase again and conclude, except that the second derivation mentions a in the context rather than a'. But $\Gamma, x : A_1, y : A_2@\theta' \vdash^{\mathsf{L}} \mathsf{refl} : a = a'$ by TRefl, so by context conversion (Lemma 5.2.27) we obtain
$$\Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a' \vdash^{\theta} b' : B$$
as needed.

– Suppose instead that a is $\langle v_1, v_2\rangle$ and $(\mathsf{pcase}_z\ \langle v_1, v_2\rangle\ \mathsf{of}\ \langle x, y\rangle \Rightarrow b) \Rrightarrow [v_1'/x][v_2'/y][\mathsf{refl}/z]b'$ by PCaseP, so $v_1 \Rrightarrow v_1'$, $v_2 \Rrightarrow v_2'$ and $b \Rrightarrow b'$.

We must show that $\Gamma \vdash^{\theta} [v_1'/x][v_2'/y][\mathsf{refl}/z]b' : B$. By the IH for $\mathcal{D}_2$, we know:
$$\Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = \langle v_1, v_2\rangle \vdash^{\theta} b' : B$$
Note that x, y and z do not appear free in B due to the assumption $\Gamma \vdash B : \star_\sigma$ of the typing rule. So the desired result will follow by substitution (Lemma 5.2.30) if we can verify that the three substituted terms have the appropriate types for the context.

We begin by observing that by the IH for $\mathcal{D}_1$ and rule PPair1, $\Gamma \vdash^{\theta} \langle v_1', v_2'\rangle : (\Sigma x : A_1 \,.\, A_2)@\theta'$. By EUnboxVal and inversion for pairs at Σ types (Lemma 5.2.51), we have $\Gamma \vdash^{\theta'} v_1' : A_1$ and $\Gamma \vdash^{\theta'} v_2' : [v_1'/x]A_2$. So one application of substitution yields:
$$\Gamma, y : [v_1'/x]A_2@\theta', z : \langle v_1', y\rangle = \langle v_1, v_2\rangle \vdash^{\theta} [v_1'/x]b' : B$$
And a second application of substitution (along with Lemma 5.2.28) yields:
$$\Gamma, z : \langle v_1', v_2'\rangle = \langle v_1, v_2\rangle \vdash^{\theta} [v_2'/y][v_1'/x]b' : B$$
By rule TRefl (and using PPair1 to construct the appropriate reduction proof), $\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : \langle v_1', v_2'\rangle = \langle v_1, v_2\rangle$, so we obtain, via a third application of the substitution lemma:
$$\Gamma \vdash^{\theta} [\mathsf{refl}/z][v_2'/y][v_1'/x]b' : B$$
Now, we may assume by the bound variable convention that x, y and z to not occur free in $v2'$ or $v1'$. So, $[\mathsf{refl}/z][v_2'/y][v_1'/x]b' = [v_1'/x][v_2'/y][\mathsf{refl}/z]b'$, concluding the case.

- Cases ENCase and ESCase are similar to EPCase.

- $$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \overset{\mathcal{F}}{\Gamma \vdash [A/x]k_2}}{\Gamma \vdash B\ A : [A/x]k_2}\ \text{TAppTLC}$$

There are two ways for $B\ A$ to take a step in the parallel reduction relation. We consider each individually.

- Suppose $B\ A \Rrightarrow B'\ A'$ by PTAppT1, so $B \Rrightarrow B'$ and $A \Rrightarrow A'$. By the IHs for $\mathcal{E}_1$ and $\mathcal{E}_2$ we have $\Gamma \vdash B' : (x : k_1) \to k_2$ and $\Gamma \vdash A' : k_1$. And by the IH for $\mathcal{F}$ (and Lemma 5.2.12), $\Gamma \vdash [A'/x]k_2$. So TAppTLC yields that $\Gamma \vdash B'\ A' : [A'/x]k_2$.

  By TReflT, $\Gamma \vdash^{\mathsf{L}} \mathsf{trefl} : A' = A$, since $A' \Rrightarrow^* A'$ and $A \Rrightarrow A'$. Thus, by TConvT, $\Gamma \vdash B'\ A' : [A/x]k_2$ as desired.

- Suppose instead that B is $\lambda x : k_1'.B_1$, A is a value, and $B\ A \Rrightarrow [A'/x]B'$ by PTBetaT, so $B_1 \Rrightarrow B'$ and $A \Rrightarrow A'$. We must show that $\Gamma \vdash [A'/x]B' : [A'/x]k_2$.

  By the IHs for $\mathcal{E}_1$ and $\mathcal{E}_2$, we have $\Gamma \vdash \lambda x : k_1'.B' : (x : k_1) \to k_2$ and $\Gamma \vdash A' : k_1$, respectively. Additionally, by Lemma 5.2.9, A' is a value. Thus, by inversion for type-level lambdas (Lemma 5.2.45), we have $\Gamma \vdash [A'/x]B' : [A'/x]k_2$, as desired.

- Case TAppDep is similar to TAppTLC.

□

Theorem 5.2.53 (Preservation)**.**

- If $\Gamma \vdash^\theta a : A$ and $a \rightsquigarrow a'$ then $\Gamma \vdash^\theta a' : A$.
- If $\Gamma \vdash A : k$ and $A \rightsquigarrow A'$ then $\Gamma \vdash A' : k$.

Proof. This is an immediate consequence of preservation for parallel reduction (Theorem 5.2.52) and the fact that $\rightsquigarrow$ is a subrelation of $\Rrightarrow$ (Lemma 5.2.18). □

5.3 Levels and Polymorphism

The interpretation's handling of arrow types requires a careful tracking of which types and kinds are polymorphic and which are monomorphic. We will define functions $\mathsf{up}(A)$ and $\mathsf{up}(k)$ to check if a type or kind "uses polymorphism". We track the use of polymorphism with the flag:

$$\pi ::= \tau \mid \sigma$$

Where τ indicates that polymorphism is not used and σ the opposite. We have an ordering, $\tau < \sigma$, and will frequently use $\mathsf{max}(\pi_1, \pi_2)$.

The definition of up is relatively simple. Informally, when we reach an arrow type we check to see if its domain is a type or a kind to determine if it is polymorphic. For all other terms, we simply check the subterms on which the interpretation will be called.

$$\boxed{\mathsf{up}(A)}$$

$$
\begin{array}{ll}
\mathsf{up}(x) & = \tau \\
\mathsf{up}((x : A) \to B) & = \mathsf{max}(\mathsf{up}(A), \mathsf{up}(B)) \\
\mathsf{up}((x : k) \to B) & = \sigma \\
\mathsf{up}(A@\theta) & = \mathsf{up}(A) \\
\mathsf{up}(A + B) & = \mathsf{max}(\mathsf{up}(A), \mathsf{up}(B)) \\
\mathsf{up}(\Sigma x : A \,.\, B) & = \mathsf{max}(\mathsf{up}(A), \mathsf{up}(B)) \\
\mathsf{up}(\mu\, x.A) & = \mathsf{up}(A) \\
\mathsf{up}(\lambda x : A.B) & = \mathsf{up}(B) \\
\mathsf{up}(\lambda x : k.B) & = \mathsf{up}(B) \\
\mathsf{up}(B\ a) & = \mathsf{up}(B) \\
\mathsf{up}(B\ A) & = \mathsf{max}(\mathsf{up}(B), \mathsf{up}(A)) \\
\mathsf{up}(\_\_) & = \tau
\end{array}
$$

$$\boxed{\mathsf{up}(k)}$$

$$
\begin{array}{ll}
\mathsf{up}(s) & = \tau \\
\mathsf{up}((x : A) \to k_2) & = \mathsf{max}(\mathsf{up}(A), \mathsf{up}(k_2)) \\
\mathsf{up}((x : k_1) \to k_2) & = \sigma
\end{array}
$$

Terms themselves never “use polymorphism” in this sense, since they can’t be used as types, but it will be convenient in the proof of a few lemmas to have a definition of $\mathsf{up}(a)$:

$$\boxed{\mathsf{up}(a)}$$

$$\mathsf{up}(a) \quad = \tau$$

Importantly, $\mathsf{up}(A)$ should only return σ for polymorphic types and kinds. In particular, if $\Gamma \vdash A : \star_\tau$, then we should have $\mathsf{up}(A) = \tau$.

Lemma 5.3.1 (Monotypes don’t “use polymorphism”)**.** If $\Gamma \vdash A : k$ and $\Gamma \vdash k$, then $\mathsf{up}(A) = \tau$.

Proof. By induction on the derivation $\mathcal{E} :: \Gamma \vdash A : k$.

Cases TVar, TNat, TEq, and TTEq are immediate by the definition of up. Cases TArrComp, TSigma, TSum, TMu and TAt are immediate by the definition of up and induction. Cases TArrPoly and TMonoPoly do not apply because k is $\star_\sigma$, so the assumption $\Gamma \vdash k$ contradicts Lemma 5.2.26. We consider the remaining cases individually:

- $$\mathcal{E} = \cfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \Gamma \vdash [A/x]k_2}{\Gamma \vdash B\ A : [A/x]k_2}\ \text{TAppTLC}$$

  By definition, $\mathsf{up}(B\ A) = \mathsf{max}(\mathsf{up}(B), \mathsf{up}(A))$. So it will be enough to show that $\mathsf{up}(B) = \mathsf{up}(A) = \tau$. The IHs for $\mathcal{E}_1$ and $\mathcal{E}_2$ give us exactly this if we can show $\Gamma \vdash (x : k_1) \to k_2$ and $\Gamma \vdash k_1$.

  By regularity (Lemma 5.2.25) and $\mathcal{D}_1$, we have $\Gamma \vdash (x : k_1) \to k_2$ as desired. By inversion, since the only rule that applies is KArrTLC, we find also $\Gamma \vdash k_1$.

- $$\mathcal{E} = \cfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : A) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash^{\mathsf{L}} a : A} \qquad \Gamma \vdash [a/x]k_2}{\Gamma \vdash B\ a : [A/x]k_2}\ \text{TAppDep}$$

  By definition, $\mathsf{up}(B\ a) = \mathsf{up}(B)$, so it will be enough to show that $\mathsf{up}(B) = \tau$. The IH for $\mathcal{E}_1$ gives us exactly this, if we can show $\Gamma \vdash (x : A) \to k_2$. But by regularity (Lemma 5.2.25) for $\mathcal{E}_1$, $\Gamma \vdash (x : A) \to k_2$ as desired.

- $$\mathcal{E} = \cfrac{\overset{\mathcal{E}_1}{\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : [b_1/x]k} \qquad \Gamma \vdash [b_2/x]k}{\Gamma \vdash A : [b_2/x]k}\ \text{TConv}$$

  We must show that $\mathsf{up}(A) = \tau$. By the IH for $\mathcal{E}_2$, it will be enough to show that $\Gamma \vdash [b_1/x]k$. By regularity (Lemma 5.2.25) and $\mathcal{E}_2$, we know that either $\Gamma \vdash [b_1/x]k$ or $[b_1/x]k = \star_\sigma$. So, it will be enough to show that $[b_1/x]k \neq \star_\sigma$.

  By assumption, we know that $\Gamma \vdash [b_2/x]k$. By Lemma 5.2.26, it therefore cannot be the case that $k = \star_\sigma$. Thus, $[b_1/x]k \neq \star_\sigma$ as desired.

- Case TConvT is similar to TConv.

□

Lemma 5.3.2 (Term substitution preserves up)**.**

- For any A and a, $\mathsf{up}(A) = \mathsf{up}([a/x]A)$.
- For any k and a, $\mathsf{up}(k) = \mathsf{up}([a/x]k)$.

Proof. By mutual induction on A and k. □

Lemma 5.3.3 (Type substitution preserves up)**.** Suppose $\Gamma \vdash B : k$ and $\Gamma \vdash k$.

- For any A, $\mathsf{up}(A) = \mathsf{up}([B/x]A)$.
- For any k, $\mathsf{up}(k) = \mathsf{up}([B/x]k)$.

Proof. By mutual induction on A and k. In the variable case for A, observe that, by Lemma 5.3.1, $\mathsf{up}(B) = \tau$. $\square$

Lemma 5.3.4 (Reduction preserves up)**.**

- If $\Gamma \vdash A : k$ and $A \Rrightarrow A'$, then $\mathsf{up}(A) = \mathsf{up}(A')$.
- If $\Gamma \vdash k$ and $k \Rrightarrow k'$, then $\mathsf{up}(k) = \mathsf{up}(k')$.

Proof. By mutual induction on the derivations of $\Gamma \vdash A : k$ and $\Gamma \vdash k$, examining the possible ways A or k can step in each case. Most cases are immediate by induction. Case TAppTLC requires Lemma 5.3.3, regularity (Lemma 5.2.25), and inversion for arrow types (Lemma 5.2.44) to see that the argument must be a monotype. $\square$

Lemma 5.3.5. For any A and B, $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(A)) = \mathsf{up}(A)$.

Proof. By definition $\mathsf{up}((x : A) \to B) = \mathsf{max}(\mathsf{up}(A), \mathsf{up}(B))$. So $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(A)) = \mathsf{min}(\mathsf{max}(\mathsf{up}(A), \mathsf{up}(B)), \mathsf{up}(A))$, and a simple case analysis shows this is $\mathsf{up}(A)$ as desired. $\square$

Lemma 5.3.6. For any A and B, $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(B)) = \mathsf{up}(B)$.

Proof. By definition, $\mathsf{up}((x : A) \to B) = \mathsf{max}(\mathsf{up}(A), \mathsf{up}(B))$. So $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(B)) = \mathsf{min}(\mathsf{max}(\mathsf{up}(A), \mathsf{up}(B)), \mathsf{up}(B))$, and a simple case analysis shows this is $\mathsf{up}(B)$ as desired. $\square$

5.4 The Interpretation

The interpretation is defined as four mutually recursive functions:

- $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho}$ is the "value" interpretation of types.
- $\mathcal{C}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho}$ is the "computation" interpretation of types.
- $\mathcal{V}_{\pi}[\![k]\!]^{j}_{\rho}$ is the "value" interpretation of kinds.
- $\mathcal{C}_{\pi}[\![k]\!]^{j}_{\rho}$ is the "computation" interpretation of kinds.

The intent is that the value interpretations describe which values belong to each type or kind, while the computation interpretations handle arbitrary terms or types. We use Ω to range over $\mathcal{V}$ and $\mathcal{C}$.

The interpretations take several arguments: a polymorphism flag π, a step count k, an environment ρ, and a term A. The "polymorphism" flag keeps track of whether the type or kind currently being interpreted is permitted to be polymorphic or not. This flag is required to make the definition well founded. It may be σ for "polymorphism allowed" or τ or "monomorphic". In the case of the type interpretations, there is also a logicality argument θ, which indicates whether we are describing terms of this type in the logical or programmatic fragments.

Environments ρ carry information about the free variables in the derivation and correspond to the derivation's context.

$$\rho ::= \emptyset \mid \rho[x \mapsto v] \mid \rho[x \mapsto (A, \mathcal{I})]$$

For each variable in the context, ρ carries one or two pieces of data. First, the interpretation always carries an expression that is intuitively the value of the variable. For a technical reason that will arise in the definition of the interpretation of function applications, we require the expressions associated with term variables to be values, while for type variables we allow any type. This term or type is used to fill the variable in when substitute the environment into another expression. For type variables, the environment also carries a "type" interpretation $\mathcal{I}$ (i.e., the kind of thing computed by the interpretation function itself). In particular, $\mathcal{I}$ will be a function from natural numbers and logicalities to set-theoretic objects. This is necessary since we will not know, at the time when we extend the environment, the step index or fragment at which the new binding will be used.

The intention is that $\mathcal{I}$ should be the interpretation of A. This may make it seem as though $\mathcal{I}$ is superfluous, since we have in A the information necessary to reconstruct it. However, keeping $\mathcal{I}$ in the context is necessary to define the interpretation as a well-founded function. Put another way, though we will always maintain the invariant that $\mathcal{I}$ is the interpretation of A, we cannot show this until after the interpretation is defined.

We define several pieces of notation relating to ρ. We write $\rho\, a$ or $\rho\,(a)$ for the term obtained by the simultaneous substitution of the values and types in ρ for the variables of a. That is, if the binding $[x \mapsto v]$ (or $[x \mapsto (A, \mathcal{I})]$) appears in ρ, then v (or A) will be substituted for all the free occurrences of x in a. The substitutions $\rho\, A$ and $\rho\, k$ are defined similarly.

5.4.1 Type of the Interpretation

If we were performing the normalization proof in a proof assistant, the interpretation would return an object in Type Theory. On paper, it is more convenient to interpret our language into Set Theory. The interpretation of a kind always returns a set of types. The interpretation of a type constructor can return different kinds of objects:

- The interpretation of a proper type is a set of terms.
- The interpretation of a type function is a set-theoretic function.

Before we can define the interpretation, we need to describe these two possibilities more formally. In particular, we will define for each kind k a set $\mathsf{Cand}(k)$ such that if $\Gamma \vdash A : k$ then $\mathsf{Cand}(k)$ describes the types of things the interpretation of interpretation of A may be. The possibilities are described by the following grammar:

$$U ::= \mathsf{UBase} \,|\, \mathsf{UTArr}\ (x : A)\ U_2 \,|\, \mathsf{UKArr}\ (x : k)\ U_2$$

This grammar describes a "universe", characterizing the types of things the interpretation may return. Roughly UBase indicates the interpretation will return a set of values, while $\mathsf{UTArr}\ (x : A)\ U_2$ and $\mathsf{UKArr}\ (x : k)\ U_2$ indicate the interpretation will return a set-theoretic function. Here, the former is for type-level functions whose arguments are terms and the latter for type-level functions whose arguments are types. We associate each kind with a universe:

$$\begin{aligned}
\mathsf{Cand}(s) &= \mathsf{UBase}\\
\mathsf{Cand}((x : A) \to k_2) &= \mathsf{UTArr}\ (x : A)\ (\mathsf{Cand}(k_2))\\
\mathsf{Cand}((x : k_1) \to k_2) &= \mathsf{UKArr}\ (x : k_1)\ (\mathsf{Cand}(k_2))
\end{aligned}$$

We will define an interpretation of these universes into set theory, formalizing this intuition. First, we introduce convenient notations to indicate sets of term and type values. We write $\mathsf{Val}^{\theta}_{\Gamma}(A)$ for the set of values of type A at logicality θ in context Γ. We write $\mathsf{Typ}_{\Gamma}(k)$ for the types of kind k in context Γ. That is:

$$\begin{aligned}
\mathsf{Val}^{\theta}_{\Gamma}(A) &= \{v \,|\, \Gamma \vdash^{\theta} v : A\}\\
\mathsf{Typ}_{\Gamma}(k) &= \{A \,|\, \Gamma \vdash A : k\}
\end{aligned}$$

Additionally, we write $\mathsf{CVal}^{\theta}(A)$ and $\mathsf{CTyp}(k)$ for the closed values and types, so that:

$$\begin{aligned} \mathsf{CVal}^{\theta}(A) = \quad & \mathsf{Val}^{\theta}_{\cdot}(A) = \quad & \{v \mid \cdot \vdash^{\theta} v : A\} \\ \mathsf{CTyp}(k) = \quad & \mathsf{Typ}_{\cdot}(k) = \quad & \{A \mid \cdot \vdash A : k\} \end{aligned}$$

Finally, we define the collections of all closed values that typecheck and all closed types that kindcheck:

$$\mathsf{VAL} = \bigcup_{A,\theta}(\mathsf{CVal}^{\theta}(A))$$

$$\mathsf{TYP} = \bigcup_{k}(\mathsf{CTyp}(k))$$

For a convenient interpretation into set theory, we overapproximate the domains of the interpretations of type-level functions. In particular, they will accept any values (or types), and no relation will be demanded between type arguments and set arguments. However, we only actually care about their behavior on the "well-formed" subset of their inputs, an idea we will make formal when we define a notion of equivalence on interpretations.

Our intuition is that, if $\Gamma \vdash A : k$, then the interpretation of A should be in the set $[\mathsf{Cand}(k)]$.

$$\begin{aligned} &[\mathsf{UBase}] && = \mathcal{P}(\mathsf{VAL}) \\ &[\mathsf{UTArr}\ (x : A)\ U_2] && = \{\emptyset\} \cup (\mathsf{VAL} \to [U_2]) \\ &[\mathsf{UKArr}\ (x : k)\ U_2] && = \{\emptyset\} \cup ((\mathsf{TYP} \times [\mathsf{Cand}(k)]^{(\theta\times\mathbb{N})}) \to [U_2]) \end{aligned}$$

Here we use the notation $[\mathsf{Cand}(k)]^{(\theta\times\mathbb{N})}$ to indicate the function space $((\{\mathsf{L},\mathsf{P}\}\times\mathbb{N}) \to [\mathsf{Cand}(k)])$. So the interpretation of a type-level function whose domain as a kind is a set-theoretic function that takes another set-theoretic function as an argument. As we will see below, this is necessary because, at the time when we interpret a type-level function, we will not know at what logicality or step-index its argument will needed.

The over-approximation of interpretation universes conveniently allows us to ignore the term and type components of kinds:

Lemma 5.4.1 (Cand universes ignore terms)**.**
For any kind k, $[\mathsf{Cand}(k)] = [\mathsf{Cand}([a/x]k)]$.

Proof. By induction on k. □

Lemma 5.4.2 (Cand universes ignore types)**.**
For any kind k, $[\mathsf{Cand}(k)] = [\mathsf{Cand}([A/x]k)]$.

Proof. By induction on k. □

Lemma 5.4.3 (Cand universes ignore environments)**.**
For any kind k and environment ρ, $[\mathsf{Cand}(k)] = [\mathsf{Cand}(\rho\, k)]$.

Proof. By induction on k. □

Lemma 5.4.4 (Cand term substitution)**.** For any a, $[a/x]\mathsf{Cand}(k) = \mathsf{Cand}([a/x]k)$.

Proof. By induction on k. □

Lemma 5.4.5 (Cand type substitution)**.** For any A, $[A/x]\mathsf{Cand}(k) = \mathsf{Cand}([A/x]k)$.

Proof. By induction on k. □

The following slightly unusual lemma is an example of the type of syntax-oriented result that would be difficult to prove with a collapsed syntax.

Lemma 5.4.6. If $\Gamma \vdash \lambda x : k_1'.B : (x : k_1) \to k_2$, then $[\mathsf{Cand}(k_1')] = [\mathsf{Cand}(k_1)]$.

Proof. By induction on the derivation $\mathcal{E} :: \Gamma \vdash \lambda x : k_1'.B : (x : k_1) \to k_2$. Due to the syntactic restrictions on the conclusion of the derivation, only three cases can apply. Case TLamTLC is immediate because k_1' and k_1 are the same.

- $$\mathcal{E} = \dfrac{\Gamma \vdash^{\mathsf{L}} a : b_1 = b_2 \qquad \overset{\mathcal{E}'}{\Gamma \vdash \lambda x : k_1'.B : [b_1/x]k} \qquad \Gamma \vdash [b_2/x]k}{\Gamma \vdash \lambda x : k_1'.B : [b_2/x]k}\ \text{TConv}$$

  Here, we know that $[b_2/x]k$ is $(x : k_1) \to k_2$ by the statement of the theorem. So, it must be the case that k is $(x : k_1'') \to k_2''$ for some k_1'' and k_2'' such that $[b_2/x]k_1'' = k_1$ and $[b_2/x]k_2'' = k_2$. The IH for $\mathcal{E}'$ therefore yields that $[\mathsf{Cand}([b_1/x]k_1'')] = [\mathsf{Cand}(k_1')]$. But because Cand universes ignore terms (Lemma 5.4.1), we know that $[\mathsf{Cand}([b_1/x]k_1'')] = [\mathsf{Cand}([b_2/x]k_1'')] = [\mathsf{Cand}(k_1)]$, concluding the case.

- Case TConvT is similar.

□

5.4.2 Definition of the Interpretation

As described above, we will define interpretations for both types and kinds. The kind interpretation is step indexed, but only so that it has a count to pass to the type interpretation. It is conceptually cleaner to consider three interpretations instead—one for monomorphic types, one for polymorphic types, and one for kinds. However, since the first two have quite a bit of overlap, we distinguish them with the polymorphism flag π instead of writing them separately.

In most of the recursive calls, the polymorphism flag has the form $\mathsf{min}(\pi, \mathsf{up}(A))$ where π is the flag that was passed in and A is the type being interpreted. This accomplishes two goals. First, the π flag needs to descend for the well foundedness of the interpretation. Second, we wish to maintain the invariant that whenever we interpret a type A, we use the polymorphism flag $\mathsf{up}(A)$.

We will use the notation $\mathcal{V}_\pi^{\{\mathsf{L},\mathsf{P}\}}[\![A]\!]_\rho^{[0..j]}$ to indicate the following function:

$$(\theta \in \{\mathsf{L},\mathsf{P}\}, i \in \mathbb{N}) \mapsto \begin{cases} \mathcal{V}_\pi^\theta[\![A]\!]_\rho^i & \text{when } i \leq j \\ \mathcal{V}_\pi^\theta[\![A]\!]_\rho^j & \text{when } i > j \end{cases}$$

This notation will be used when extending the context, since we will not know at what logicality or step index the new binding will be used. We will, however, know that the step index is no larger than the current step index, justifying the maximum bound in this definition (and helping to keep the definition of the interpretation well founded, later).

We are now prepared to define the interpretation. We begin with the value interpretation of types. This is a set-theoretic function defined by recursion. Intuitively, the metric which descends in every recursive call is the lexicographically ordered triple (π, j, A). We will give a more precise proof that the definition is well founded in Section 5.5.

$$\mathcal{V}_\pi^\theta\,[\![x]\!]_\rho^j = \begin{cases} \mathcal{I}(\theta, j) & \text{when } \rho = \rho_1[x \mapsto (V, \mathcal{I})]\,\rho_2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathcal{V}_\pi^{\mathsf{L}}\,[\![(x:A) \to B]\!]_\rho^j = {} & \\
& \{\lambda x{:}A'.b \mid \quad \cdot \vdash^{\mathsf{L}} \lambda x{:}A'.b : \rho\,((x:A) \to B) \\
& \qquad \text{and } \forall i \leq j,\ \forall v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]_\rho^i, \\
& \qquad\quad [v/x]b \in \mathcal{C}^{\mathsf{L}}_{\mathsf{min}(\pi,\mathsf{up}(B))}[\![B]\!]^i_{\rho[x\mapsto v]}\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}_\pi^{\mathsf{P}}\,[\![(x:A) \to B]\!]_\rho^j = {} & \\
& \{\lambda x{:}A'.b \mid \quad \cdot \vdash^{\mathsf{P}} \lambda x{:}A'.b : \rho\,((x:A) \to B) \\
& \qquad \text{and } \forall i < j,\ \forall v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]_\rho^i, \\
& \qquad\quad [v/x]b \in \mathcal{C}^{\mathsf{P}}_{\mathsf{min}(\pi,\mathsf{up}(B))}[\![B]\!]^i_{\rho[x\mapsto v]}\} \\
& \cup \{\mathsf{rec}\, f\,(x{:}A').b \mid \quad \cdot \vdash^{\mathsf{P}} \mathsf{rec}\, f\,(x{:}A').b : \rho\,((x:A) \to B) \\
& \qquad \text{and } \forall i < j,\ \forall v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]_\rho^i, \\
& \qquad\quad [v/x][\mathsf{rec}\, f\,(x{:}A').b/f]b \in \mathcal{C}^{\mathsf{P}}_{\mathsf{min}(\pi,\mathsf{up}(B))}[\![B]\!]^i_{\rho[x\mapsto v]}\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}_\sigma^{\mathsf{L}}\,[\![(x:k) \to B]\!]_\rho^j = {} & \\
& \{\lambda x{:}k'.b \mid \quad \cdot \vdash^{\mathsf{L}} \lambda x{:}k'.b : \rho\,((x:k) \to B) \\
& \qquad \text{and } \forall i \leq j,\ \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]_\rho^i, \\
& \qquad\quad [V/x]b \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^i_{\rho[x\mapsto(V,\mathcal{V}_\tau^{\{\mathsf{L},\mathsf{P}\}}[\![V]\!]_\emptyset^{[0..i]})]}\}
\end{aligned}$$

$$\mathcal{V}_\tau^{\mathsf{L}}\,[\![(x:k) \to B]\!]_\rho^j = \emptyset$$

$$
\begin{aligned}
&\mathcal{V}^{\mathsf{P}}_{\sigma}\,[\![(x:k)\to B]\!]^{j}_{\rho} = \\
&\qquad\{\lambda x\!:\!k'.b \mid \quad \cdot \vdash^{\mathsf{P}} \lambda x\!:\!k'.b : \rho\,((x:k)\to B) \\
&\qquad\qquad\qquad \text{and } \forall i<j,\ \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho}, \\
&\qquad\qquad\qquad\quad [V/x]b \in \mathcal{C}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}\} \\
&\qquad\cup\{\mathsf{rec}\,f\,(x\!:\!k').b \mid \quad \cdot \vdash^{\mathsf{P}} \mathsf{rec}\,f\,(x\!:\!k').b : \rho\,((x:A)\to B) \\
&\qquad\qquad\qquad \text{and } \forall i<j,\ \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho}, \\
&\qquad\qquad\qquad\quad [v/x][\mathsf{rec}\,f\,(x\!:\!k').b/f]b \in \mathcal{C}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}^{\mathsf{P}}_{\sigma}\,[\![(x:k)\to B]\!]^{j}_{\rho} &= \emptyset \\
\mathcal{V}^{\theta}_{\pi}\,[\![A@\theta']\!]^{j}_{\rho} &= \mathcal{V}^{\theta'}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]^{j}_{\rho} \\
\mathcal{V}^{\theta}_{\pi}\,[\![\mathsf{Nat}]\!]^{j}_{\rho} &= \{\mathsf{S}^{n}\,\mathsf{Z} \mid n \in \mathbb{N}\} \\
\mathcal{V}^{\theta}_{\pi}\,[\![A+B]\!]^{j}_{\rho} &= \quad \{\mathsf{inl}\,v \mid \cdot \vdash \rho\,(A+B) : \star_{\sigma} \text{ and } v \in \mathcal{V}^{\theta}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]^{j}_{\rho}\} \\
&\quad\ \cup \{\mathsf{inl}\,v \mid \cdot \vdash \rho\,(A+B) : \star_{\sigma} \text{ and } v \in \mathcal{V}^{\theta}_{\mathsf{min}(\pi,\mathsf{up}(B))}[\![B]\!]^{j}_{\rho}\} \\
\mathcal{V}^{\theta}_{\pi}\,[\![\Sigma x\!:\!A\,.\,B]\!]^{j}_{\rho} &= \{\langle v_1,v_2\rangle \mid \cdot \vdash \rho\,(\Sigma x\!:\!A\,.\,B) : \star_{\sigma} \text{ and } v_1 \in \mathcal{V}^{\theta}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]^{j}_{\rho} \\
&\qquad\qquad \text{and } v_2 \in \mathcal{V}^{\theta}_{\mathsf{min}(\pi,\mathsf{up}(B))}[\![B]\!]^{j}_{\rho[x\mapsto v_1]}\} \\
\mathcal{V}^{\theta}_{\pi}\,[\![\mu\,x.A]\!]^{j}_{\rho} &= \{\mathsf{roll}\,v \mid \cdot \vdash^{\theta} \mathsf{roll}\,v : \rho\,(\mu\,x.A) \\
&\qquad \text{and } \forall i<j, v \in \mathcal{V}^{\theta}_{\mathsf{min}(\pi,\mathsf{up}([\mu\,x.A/x]A))}[\![[\mu\,x.A/x]A]\!]^{i}_{\rho}\} \\
\mathcal{V}^{\theta}_{\pi}\,[\![a_1=a_2]\!]^{j}_{\rho} &= \{\mathsf{refl} \mid \cdot \vdash \rho\,(a_1=a_2) : \star_{\tau} \text{ and } \exists b, \rho\,a_1 \Rrightarrow^{*} b \text{ and } \rho\,a_2 \Rrightarrow^{*} b\} \\
\mathcal{V}^{\theta}_{\pi}\,[\![A_1=A_2]\!]^{j}_{\rho} &= \{\mathsf{refl} \mid \cdot \vdash \rho\,(A_1=A_2) : \star_{\tau} \text{ and } \exists b, \rho\,A_1 \Rrightarrow^{*} B \text{ and } \rho\,A_2 \Rrightarrow^{*} B\} \\
\mathcal{V}^{\theta}_{\pi}\,[\![\lambda x:k_1.B]\!]^{j}_{\rho} &= (A \in \mathsf{TYP}, \mathcal{I} \in [\mathsf{Cand}(k_1)]^{(\theta\times\mathbb{N})}) \mapsto \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x\mapsto(A,\mathcal{I})]} \\
\mathcal{V}^{\theta}_{\pi}\,[\![\lambda x:A.B]\!]^{j}_{\rho} &= (v \in \mathsf{VAL}) \mapsto \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x\mapsto v]} \\
\mathcal{V}^{\theta}_{\pi}\,[\![B\ A]\!]^{j}_{\rho} &= \begin{cases} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho}\,(\rho\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho}) \\ \qquad \text{If } \rho\,A \rightsquigarrow^{*} V \\ \qquad\quad \text{and } (\rho\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho}) \text{ is in the domain of } \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho} \\ \emptyset \quad \text{otherwise} \end{cases} \\
\mathcal{V}^{\theta}_{\pi}\,[\![B\ a]\!]^{j}_{\rho} &= \begin{cases} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho}\,v & \text{If } \rho\,a \rightsquigarrow^{*} v \text{ and } v \text{ is in the domain of } \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

The definition above is primarily a standard extension of the interpretation from Chapter 4 to handle polymorphism and type-level computation, in the style of term models for languages like the Calculus of Constructions [64, 40, 26]. A few cases merit additional attention.

The case for variables is relatively unsurprising, recalling that the environment ρ is intended to contain, for each type variable, a function from step -indices and logicalities to the interpretation of the corresponding type at those arguments.

The interpretation of functions whose domain and range are both types is identical to the interpretation of functions from Section 4.3.1, except that PCC$^\theta$ lacks natural-number recursion so there are fewer cases and that we have added the polymorphism flag π. As described above, when recursively interpreting the type A with an input flag of π, we use the flag $\mathsf{min}(\pi, \mathsf{up}(A))$. This ensures that the flag is descending.

The situation is slightly more complicated in the case for polymorphic function types. First, we have four cases—one for each combination of the logicality θ and the polymorphism flag π. A polymorphic function is in the interpretation if it "sends related arguments to related results". But, since the argument to the function is a type, when we extend the environment with the argument we must provide its interpretation. Since the argument type is essentially arbitrary, this recursive call would break the well foundedness of the interpretation if it were not for the polymorphism flag. So, we only interpret polymorphic function types if the input polymorphism flag is σ, and in the recursive call for the function's argument we switch the flag to τ, ensuring well foundedness.

Type-level functions are interpreted as set-theoretic functions. When the function's domain is a kind, its set-theoretic interpretation takes as arguments both a type and function which, given a step index and logicality, produces interpretations of the right form. Intuitively, this function should always return the interpretation of the type argument at the appropriate index. However, stating that explicitly would break the well foundedness of the interpretation. This case of the interpretation is the reason we have annotated the domains of functions in PCC$^\theta$. Without knowing the domain of the type-level function in the system, there is no way to state an appropriate domain for its interpretation in type theory. An alternative would be to define the interpretation by induction on kinding derivations, rather than types. We will describe some problems with this approach in Chapter 6.

There are two cases that handle applications of type-level functions. One for normal type-level computation, and one for dependently-typed functions. In both cases, we check whether the function's argument terminates and return the empty set as a "dummy" answer if not. When we prove the fundamental theorem of the interpretation we'll know that the argument must terminate, so the dummy case will not arise.

In the case of a dependent application, the argument a evaluates to a value v and we simply apply the interpretation of the function to v. In the case of type-level computation, the argument is a type A which evaluates to V. Here we apply the interpretation of the function to A instead of V. The reason has to do with the ordering of two lemmas we will soon prove about the interpretation. The first (Lemma 5.7.1) says that interpreting an open type with an environment yields the same result as if we first substitute in the environment and interpret the resulting close type. The second (Lemma 5.7.4) says that if $A \Rrightarrow A'$, then A and A' have the same interpretation. The second lemma implies that it does not matter whether we apply the interpretation of the function to A or V. However, we need to prove the other lemma first, and for it we must know that the function is applied to a type and the interpretation of that type. This is the reason we allow non-values in the environment, in the case of types.

The remaining cases of the value interpretation of types closely mirror the interpretations from our previous proofs. We now define the computation interpretation for types, which is similarly unsurprising:

$$\begin{aligned}
\mathcal{C}^{\mathsf{L}}_{\pi}\,[\![A]\!]^{j}_{\rho} &= \{a \,|\, \cdot \vdash^{\mathsf{L}} a : \rho\, A \text{ and } a \rightsquigarrow^{*} v \in \mathcal{V}^{\mathsf{L}}_{\pi}[\![A]\!]^{j}_{\rho}\}\\
\mathcal{C}^{\mathsf{P}}_{\pi}\,[\![A]\!]^{j}_{\rho} &= \{a \,|\, \cdot \vdash^{\mathsf{L}} a : \rho\, A \text{ and } \forall i \le j,\ \text{if } a \rightsquigarrow^{j} v,\ \text{then } v \in \mathcal{V}^{\mathsf{P}}_{\pi}[\![A]\!]^{(j-i)}_{\rho}\}
\end{aligned}$$

We must also define an interpretation of kinds. Since there are no kind-level functions, this interpretation will always return a set of type values. Since the system does not include a distinction between logical and programmatic types, no logicality argument is needed. As in the type interpretation, we use the polymorphism flag to handle function types whose domain is a kind.

$$\begin{aligned}
\mathcal{V}_{\pi}\,[\![s]\!]^{j}_{\rho} &= \{V \,|\, \cdot \vdash V : s\}\\
\mathcal{V}_{\pi}\,[\![(x : A) \to k_2]\!]^{j}_{\rho} &= \\
&\quad \{\lambda x : A'.B \,|\, \quad \cdot \vdash \lambda x : A'.B : \rho\,((x : A) \to k_2)\\
&\qquad \text{and } \forall i \le j,\ \forall v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{min}(\pi,\mathsf{up}(A))}[\![A]\!]^{i}_{\rho},\\
&\qquad\quad [v/x]B \in \mathcal{C}_{\mathsf{min}(\pi,\mathsf{up}(k_2))}[\![k_2]\!]^{i}_{\rho[x\mapsto v]}\}\\
\mathcal{V}_{\sigma}\,[\![(x : k_1) \to k_2]\!]^{j}_{\rho} &= \\
&\quad \{\lambda x : k_1'.B \,|\, \quad \cdot \vdash \lambda x : k_1'.B : \rho\,((x : k_1) \to k_2)\\
&\qquad \text{and } \forall i \le j,\ \forall V \in \mathcal{V}_{\mathsf{up}(k_1)}[\![k_1]\!]^{i}_{\rho},\\
&\qquad\quad [V/x]B \in \mathcal{C}_{\mathsf{up}(k_2)}[\![k_2]\!]^{i}_{\rho[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}\}\\
\mathcal{V}_{\tau}\,[\![(x : k_1) \to k_2]\!]^{j}_{\rho} &= \emptyset
\end{aligned}$$

$$\mathcal{C}_{\pi}\,[\![k]\!]^{j}_{\rho} = \{A \,|\, \cdot \vdash A : \rho\, k \text{ and } A \rightsquigarrow^{*} V \in \mathcal{V}_{\pi}[\![k]\!]^{j}_{\rho}\}$$

5.5 Basic Facts About the Interpretation and Environments

Predictably, a wide variety of (mostly uninteresting) lemmas are needed for the main proof of the interpretation's soundness. In this section, we consider the basic properties of the preceding definitions. We reserve the two most interesting lemmas about the interpretation for Section 5.7.

Theorem 5.5.1 (The interpretation is well-defined)**.** The above definitions specify (well-founded) functions.

Proof. Intuitively, the interpretation is well-defined by lexicographic induction on the triple $(\pi, j, A/k)$. Here, we write A/k to indicate the type or kind argument to the appropriate interpretation, which is sensible since types and kinds are mutually defined. We could also model this as nested recursive functions, where at one level we have two mutually recursive definitions.

However, it is also the case that the computational interpretation is allowed to call the value interpretation at the same arguments it was passed. Thus, instead we proceed by induction on the quadruple $(\pi, j, A/k, \Omega)$ where Ω is either $\mathcal{C}$ or $\mathcal{V}$ with $\mathcal{V} < \mathcal{C}$.

Simple inspection of each case of the definitions above reveals that this metric always descends in a recursive call. $\square$

Lemma 5.5.2 (Environment substitution preserves head forms)**.** If $\mathsf{hd}\,(A) = hf$, then $\mathsf{hd}\,(\rho\, A) = hf$ or any ρ.

Proof. By examination of the definition of head forms. $\square$

The next two lemmas say that the types in the interpretation of a non-polymorphic kind should not be polymorphic. They are the semantic equivalent of Lemma 5.3.1.

Lemma 5.5.3. If $\mathcal{D} :: \Gamma \vdash k$ and $A \in \mathcal{C}_\pi[\![k]\!]^j_\rho$, then $\mathsf{up}(A) = \tau$.

Proof. By the definition of the interpretation, we have $\cdot \vdash A : \rho\, k$. By Lemma 5.3.1, it will therefore be enough to show that $\cdot \vdash \rho\, k$. By regularity (Lemma 5.2.25) and the derivation of $\cdot \vdash A : \rho\, k$, we have that either $\cdot \vdash \rho\, k$ or $\rho\, k = \star_\sigma$. But if $\rho\, k = \star_\sigma$, then $k = \star_\sigma$, contradicting the assumption that $\Gamma \vdash k$ (by Lemma 5.2.26). $\square$

Lemma 5.5.4. If $\mathcal{D} :: \Gamma \vdash k$ and $V \in \mathcal{V}_\pi[\![k]\!]^j_\rho$, then $\mathsf{up}(V) = \tau$.

Proof. Consider the possible cases of k:

- $k = s$.

  Then by $\mathcal{D}$ and Lemma 5.2.26, s is $\star_\tau$. By the definition of the interpretation, $\cdot \vdash V : \star_\tau$, so by Lemma 5.3.1, $\mathsf{up}(V) = \tau$ as desired.

- $k = (x : A) \to k_2$.

  By the definition of the interpretation, we have some derivation of $\cdot \vdash V : \rho\,((x : A) \to k_2)$. Since $\rho\,((x : A) \to k_2) \neq \star_\sigma$, regularity (Lemma 5.2.25) gives us that $\cdot \vdash \rho\,((x : A) \to k_2)$. So, by Lemma 5.3.1, $\mathsf{up}(V) = \tau$ as desired.

- $k = (x : k_1) \to k_2$.

  Similar to the previous case.

□

Many of the properties we'd like to prove about the interpretation are not obvious in the variable case, because we use an arbitrary set from the environment. To handle this, for each lemma we define a new judgement that captures the idea that each binding in the environment has the properties we need. It is tempting to combine these judgements into one larger judgement that captures every property we need to know about the environment at once. However, such a judgement would be difficult to work with, because when extending the environment we will typically only be able to verify that our extension maintains the property we are currently working with.

For the next lemma, we will need to know that, for each type variable $x : k$, the environment contains a mapping $[x \mapsto (A, \mathcal{I})]$ such that $\mathcal{I}(\theta, j) \in [\mathsf{Cand}(k)]$ for any θ and j. We formalize this in the relation $\mathsf{EnvCand}(\rho, \Gamma)$.

$$\frac{}{\mathsf{EnvCand}(\emptyset, \cdot)}\ \textsc{ECandN} \qquad \frac{\mathsf{EnvCand}(\rho, \Gamma)}{\mathsf{EnvCand}(\rho[x \mapsto v], (\Gamma, x : A))}\ \textsc{ECandCT}$$

$$\frac{\begin{array}{c}\mathsf{EnvCand}(\rho, \Gamma)\\ \forall \theta\, j, \mathcal{I}(\theta, j) \in [\mathsf{Cand}(k)]\end{array}}{\mathsf{EnvCand}(\rho[x \mapsto (A, \mathcal{I})], (\Gamma, x : k))}\ \textsc{ECandCK}$$

Lemma 5.5.5 (The interpretation respects $\textsc{Cand}$)**.** If $\mathcal{E} :: \Gamma \vdash A : k$ and $\mathsf{EnvCand}(\rho, \Gamma)$ then $\mathcal{V}_\pi^\theta[\![A]\!]_\rho^j \in [\mathsf{Cand}(k)]$.

Proof. By induction on $\mathcal{E}$.

- $\mathcal{E} = \dfrac{(x : k) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : k}\ \textsc{TVar}$

  In this case, we know from the definitions of the interpretation and $\mathsf{EnvCand}(\rho, \Gamma)$ that $\rho = \rho_1[x \mapsto (V, \mathcal{I})]\,\rho_2$ and that, for any θ and j, $\mathcal{I}(\theta, j) \in [\mathsf{Cand}(k)]$. But by the definition of the interpretation, $\mathcal{V}_\pi^\theta[\![x]\!]_\rho^j = \mathcal{I}(\theta, j)$, concluding the case.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash A : s} \qquad \overset{\mathcal{E}_2}{\Gamma, x : A \vdash B : s} \qquad \mathsf{Mob}\,(A)}{\Gamma \vdash (x : A) \to B : s}\ \textsc{TArrComp}$

  In this case, $[\mathsf{Cand}(k)] = [\mathsf{Cand}(s)] = [\mathsf{UBase}] = \mathsf{VAL}$. So, it will be enough to show that $\mathcal{V}_\pi^\theta[\![A]\!]_\rho^j$ is a set of values that typecheck, which is immediate by the definition of the interpretation.

- Cases $\textsc{TArrPoly}$, $\textsc{TMonoPoly}$, $\textsc{TNat}$, $\textsc{TSigma}$, $\textsc{TSum}$, $\textsc{TMu}$, $\textsc{TEq}$, $\textsc{TEqT}$, and $\textsc{TAt}$ are similar to the previous case.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma, x : k_1 \vdash B : k_2} \qquad \Gamma \vdash (x : k_1) \to k_2}{\Gamma \vdash \lambda x : k_1 . B : (x : k_1) \to k_2}$ TLamTLC

  We must show:

$$\begin{aligned}\mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1 . B]\!]^{j}_{\rho} \in [\mathsf{Cand}((x : k_1) \to k_2)] \\ = [\mathsf{UKArr}\ (x : k_1)\ (\mathsf{Cand}(k_2))] \\ = \{\emptyset\} \cup ((\mathsf{TYP} \times [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \to [\mathsf{Cand}(k_2)])\end{aligned}$$

  By definition, we have:

$$\mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1 . B]\!]^{j}_{\rho} \qquad = (A \in \mathsf{TYP}, \mathcal{I} \in [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \mapsto \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto (A, \mathcal{I})]}$$

  This function has the required domain by definition. The IH for $\mathcal{D}_1$ gives us that $\mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto (V, \mathcal{I})]} \in [\mathsf{Cand}(k_2)]$ (observing that the requirements to extend EnvWt are trivially satisfied by the function's domain).

- Case TLamDep is similar to (but simpler than) the previous case.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \Gamma \vdash [A/x]k_2}{\Gamma \vdash B\ A : [A/x]k_2}$ TAppTLC

  We must show that $\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j}_{\rho} \in [\mathsf{Cand}([A/x]k_2)]$. By definition, we have:

$$\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j}_{\rho} \quad = \begin{cases} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho}\ (\rho\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho}) \\ \quad \text{If } \rho\, A \rightsquigarrow^{*} V \\ \quad\quad \text{and } (\rho\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho}) \text{ is in the domain of } \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho} \\ \emptyset \quad \text{otherwise} \end{cases}$$

  The IH for $\mathcal{D}_1$ gives us that:

$$\begin{aligned}\mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho} \in [\mathsf{Cand}((x : k_1) \to k_2)] \\ = [\mathsf{UKArr}\ (x : k_1)\ (\mathsf{Cand}(k_2))] \\ = \{\emptyset\} \cup ((\mathsf{TYP} \times [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \to [\mathsf{Cand}(k_2)])\end{aligned}$$

  Now, if the latter case of the definition of $\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j}_{\rho}$ applies, the result is immediate because $\emptyset$ is in $[U]$ for any U. So, we may assume that $\mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho}$ is a function and that $(V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho})$ is in its domain. It follows from the IH that $\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j}_{\rho} \in [\mathsf{Cand}(k_2)]$.

  By Lemma 5.4.2, $[\mathsf{Cand}(k_2)] = [\mathsf{Cand}([A/x]k_2)]$, concluding the case.

- Case TAppDep is similar to the previous case.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash^{\mathsf{L}} b : B_1 = B_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : [B_1/x]k} \qquad \Gamma \vdash [B_2/x]k}{\Gamma \vdash A : [B_2/x]k}$ TConvT

  We must show that $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho} \in [\mathsf{Cand}([B_2/x]k)]$. By the IH for $\mathcal{E}_2$, we have $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho} \in [\mathsf{Cand}([B_1/x]k)]$. By Lemma 5.4.2, $[\mathsf{Cand}([B_1/x]k)] = [\mathsf{Cand}(k)] = [\mathsf{Cand}([B_2/x]k)]$, concluding the case.

- Case TConv is similar to the previous case.

□

For the next lemma, we will need to know that each term or type in the environment has the expected type or kind and that types in the environment are normalizing. We define a new judgement which captures these facts.

$$\frac{}{\mathsf{EnvTyp}(\emptyset, \cdot)}\ \text{ETypN} \qquad \frac{\begin{array}{c}\mathsf{EnvTyp}(\rho, \Gamma)\\ \cdot \vdash^{\mathsf{L}} v : \rho\, A\end{array}}{\mathsf{EnvTyp}(\rho[x \mapsto v], (\Gamma, x : A))}\ \text{ETypCT}$$

$$\frac{\begin{array}{l}\mathsf{EnvTyp}(\rho, \Gamma)\\ \cdot \vdash A : \rho\, k\\ A \Rrightarrow^{*} V\end{array}}{\mathsf{EnvTyp}(\rho[x \mapsto (A, \mathcal{I})], (\Gamma, x : k))}\ \text{ETypCK}$$

We begin with two basic facts about well-typed environments:

Lemma 5.5.6 (Environment substitution preserves term values)**.** If $\Gamma \vdash^{\theta} v : A$ and $\mathsf{EnvTyp}(\rho, \Gamma)$, then $\rho\, v$ is a value.

Proof. By induction on the typing derivation. In the variable case, the definition of $\mathsf{EnvTyp}(\rho, \Gamma)$ ensures $\rho\, x$ is a value. □

Lemma 5.5.7 (Environment substitution)**.** If $\mathsf{EnvTyp}(\rho, \Gamma)$, then the following all hold:

- If $\Gamma \vdash^{\theta} a : A$ then $\cdot \vdash^{\theta} \rho\, a : \rho\, A$.
- If $\Gamma \vdash A : k$ then $\cdot \vdash \rho\, A : \rho\, k$.
- If $\Gamma \vdash k$ then $\cdot \vdash \rho\, k$.

Proof. By induction on the proof of $\mathsf{EnvTyp}(\rho, \Gamma)$, using Lemmas 5.2.30 and 5.2.32.
□

Another convenient fact about well-typed environments is that substituting them into a type or kind does not change whether it uses polymorphism:

Lemma 5.5.8. Suppose $\mathsf{EnvTyp}(\rho_1\,\rho_2\,\rho_3, \Gamma)$ and $\vdash \Gamma$.

- For any A, $\mathsf{up}(A) = \mathsf{up}(\rho_2\, A)$.
- For any k, $\mathsf{up}(k) = \mathsf{up}(\rho_2\, k)$.

Proof. By induction on ρ_2.

For value bindings, the result is immediate by induction and Lemma 5.3.2.

When ρ_2 has the form $\rho_2'[x \mapsto (B, \mathcal{I})]$, observe that $\mathsf{EnvWT}(\rho_1\,\rho_2\,\rho_3, \Gamma)$ means that $\cdot \vdash B : \rho_1\,\rho_2'\,k'$ for some k' such that $\Gamma = \Gamma_1, x : k', \Gamma_2$. The proof of $\vdash \Gamma$ ensures that $\Gamma_1 \vdash k'$ and thus, by environment substitution (Lemma 5.5.7, $\cdot \vdash \rho_1\,\rho_2'\,k'$. By Lemma 5.3.3, $\mathsf{up}([B/x]\rho_2'\,A) = \mathsf{up}(\rho_2'\,A)$. The result then follows by induction. □

Lemma 5.5.9 (The interpretation models mobility)**.** If $\mathsf{Mob}\,(A)$ then $\Omega^{\mathsf{L}}_{\pi}[\![A]\!]^{j}_{\rho} = \Omega^{\mathsf{P}}_{\pi}[\![A]\!]^{j}_{\rho}$.

Proof. By induction on the pair (A, Ω), so that the IH will be available for smaller types A or when A remains constant but Ω is $\mathcal{C}$ and we want the IH for $\mathcal{V}$.

When Ω is C, the result holds immediately by the definition of the interpretation and the induction hypothesis for $(A, \mathcal{V})$.

In the case when Ω is $\mathcal{V}$, we consider the possible forms of A. Most cases are ruled out because there is no derivation that the appropriate form is mobile. The cases for Nat, $a = b$, $A = B$, and $A@\theta$ are immediate because the definition of the interpretation does not mention the given logicality. Only two cases remain:

- Suppose $A = \Sigma x : A_1 \,.\, A_2$. We know $\mathsf{Mob}\,(A_1)$ and $\mathsf{Mob}\,(A_2)$ by inversion on the proof of $\mathsf{Mob}\,(\Sigma x : A_1 \,.\, A_2)$. The result follows immediately by induction and the definition of the interpretation.
- The case where $A = A_1 + A_2$ is similar to the previous case.

□

Lemma 5.5.10 (Interpretation Weakening and Strengthening)**.** Suppose $\mathsf{dom}(\rho_2) \cap \mathsf{dom}(\rho_1\,\rho_3) = \emptyset$ and let some type A and kind k be given.

- If $\mathsf{fvs}(A) \subseteq \mathsf{dom}(\rho_1\,\rho_2)$ then $\Omega^{\theta}_{\pi}[\![A]\!]^{j}_{\rho_1\,\rho_3} = \Omega^{\theta}_{\pi}[\![A]\!]^{j}_{\rho_1\,\rho_2\,\rho_3}$.
- If $\mathsf{fvs}(k) \subseteq \mathsf{dom}(\rho_1\,\rho_2)$ then $\Omega_{\pi}[\![k]\!]^{j}_{\rho_1\,\rho_3} = \Omega_{\pi}[\![k]\!]^{j}_{\rho_1\,\rho_2\,\rho_3}$.

Proof. By induction on the (lexicographically-ordered) triple $(j, A/k, \Omega)$. Note that we leave π and ρ_2 general, so that the IH will apply for any instantiation of these arguments.

If Ω is $\mathcal{C}$, the theorem is immediate by the definition of the interpretation and the induction hypothesis for $\mathcal{V}$.

If Ω is $\mathcal{V}$, we consider the possible forms of A and k individually. Most cases are immediate by the definition of the interpretation and an invocation of the appropriate IH. Some cases, like TEqT, are slightly more complicated because the environment is also substituted into a term, type or kind in a typing or reduction assumption. In these cases the relevant term type or kind is always a subexpression of A (or k), so it is sufficient to observe that substituting $\rho_1\,\rho_3$ into the expression is the same as substituting in $\rho_1\,\rho_2\,\rho_3$, since the variables in ρ_2's domain do not appear free in A (or k). We spell out the variable case in slightly greater detail:

- Case: $A = x$.

  By assumption, x cannot be in the domain of ρ_2. Thus, an interpretation for x appears in $\rho_1\,\rho_3$ iff it appears in $\rho_1\,\rho_2\,\rho_3$, and it is the same in either environment.

$\square$

5.5.1 Universe-Indexed Properties of the Interpretation

Several properties about the interpretation which were quite straightforward to state and prove in a system without type-level computation become somewhat more complicated in the present setting. For example, in the case of the interpretation from Chapter 3, it's quite easy to see that if v is in the interpretation of A, then $\cdot \vdash^\theta v : A$. To prove a similar lemma for PCC$^\theta$, we must generalize its statement to account for the possibility that the interpretation of A is set-theoretic function rather than a set of values.

We now define a more general version of this property by recursion on universes.

Definition 5.5.11. We define the property $\mathsf{WT}^\theta(U, A, \mathcal{I})$ as follows:

- $\mathsf{WT}^\theta(\mathsf{UBase}, A, \mathcal{I})$ iff $\mathcal{I} \in \mathsf{VAL}$ and, if $v \in \mathcal{I}$, then $\cdot \vdash^\theta v : A$.
- $\mathsf{WT}^\theta(\mathsf{UTArr}\ (x : B)\ U, A, \mathcal{I})$ iff $\mathcal{I} = \emptyset$, or
  - $\mathcal{I} : (\mathsf{VAL} \to [U])$, and
  - $\forall v \in \mathsf{CVal}^{\mathsf{L}}(B)$, $\mathsf{WT}^\theta([v/x]U, A\ v, \mathcal{I}\ v)$
- $\mathsf{WT}^\theta(\mathsf{UKArr}\ (x : k)\ U, A, \mathcal{I})$ iff $\mathcal{I} = \emptyset$, or
  - $\mathcal{I} : (\mathsf{TYP} \times [\mathsf{Cand}(k)]^{(\theta\times\mathbb{N})}) \to [U]$
  - and,

$$\begin{aligned} &\forall B \in \mathsf{CTyp}(k), \forall \mathcal{I}' \in [\mathsf{Cand}(k)]^{(\theta\times\mathbb{N})},\\ &\quad B \Rrightarrow^* V\\ &\to \forall j, \forall \theta', \mathsf{WT}^{\theta'}(\mathsf{Cand}(k), B, \mathcal{I}'(\theta', j))\\ &\to \mathsf{WT}^\theta([B/x]U, A\ B, \mathcal{I}(B, \mathcal{I}'))\end{aligned}$$

It is not immediately obvious that this is well-defined. To see that it is, we define two size functions:

$$\begin{aligned}
&\mathsf{sizek}(s) &&= 0\\
&\mathsf{sizek}((x:A)\to k_2) &&= 1+\mathsf{sizek}(k_2)\\
&\mathsf{sizek}((x:k_1)\to k_2) &&= 1+\mathsf{sizek}(k_1)+\mathsf{sizek}(k_2)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sizeU}(\mathsf{UBase}) &&= 0\\
&\mathsf{sizeU}(\mathsf{UTArr}\ (x:A)\ U_2) &&= 1+\mathsf{sizeU}(U_2)\\
&\mathsf{sizeU}(\mathsf{UKArr}\ (x:k_1)\ U_2) &&= 1+\mathsf{sizek}(k_1)+\mathsf{sizeU}(U_2)
\end{aligned}$$

It is simple to show that $\mathsf{sizek}(k) = \mathsf{sizeU}(\mathsf{Cand}(k))$ and that substituting a term or type into kinds or universes does not change their sizes. We will use these facts implicitly when performing induction on the size of a kind of universe in the future, to avoid getting bogged down in notation and lemmas. With these lemmas in hand, we see that the above definition for $\mathsf{WT}^{\theta}(U, A, \mathcal{I})$ is well founded because $\mathsf{sizeU}(U)$ decreases in each recursive call.

It's important to note that, unlike the definition of $[U]$, the definition of $\mathsf{WT}^{\theta}(U, A, \mathcal{I})$ makes use of the types and kinds within U. For this reason, we will be careful about free variables when reasoning about a universe.

Before we can prove that the interpretation always produces sets in the WT relation, we must prove a few properties of the relation itself.

Lemma 5.5.12. If $\mathsf{WT}^{\theta}(\mathsf{Cand}(k), A, \mathcal{I})$, then $\mathcal{I} \in [\mathsf{Cand}(k)]$.

Proof. By examining the definition of WT for each of the three possible forms for k. $\square$

Lemma 5.5.13. If $\cdot \vdash A : k$ and $A \Rrightarrow A'$ then
$\mathsf{WT}^{\theta}(\mathsf{Cand}(k), A, \mathcal{I}) \leftrightarrow \mathsf{WT}^{\theta}(\mathsf{Cand}(k), A', \mathcal{I})$.

Proof. By induction on $\mathsf{sizek}(k)$. Consider the possible cases of k.

- Suppose k is s.

  We must show that $v \in \mathsf{CVal}^{\theta}(A) \leftrightarrow v \in \mathsf{CVal}^{\theta}(A')$. Observe that $\cdot \vdash A' : k$ by preservation (Theorem 5.2.52). So $\cdot \vdash^{\mathsf{L}} \mathsf{refl} : A = A'$, and the result follows by EConvT.

- Suppose k is $(x : B_1) \to k_2$, so $\mathsf{Cand}(k)$ is $\mathsf{UTArr}\ (x : B_1)\ (\mathsf{Cand}(k_2))$.

  Consider first the $\to$ direction. If $\mathcal{I} = \emptyset$, the result is immediate, so suppose instead that:

  - $\mathcal{I} : (\mathsf{VAL} \to [\mathsf{Cand}(k_2)])$, and

– $\forall v \in \mathsf{CVal}^{\mathsf{L}}(B_1)$, $\mathsf{WT}^{\theta}([v/x]\mathsf{Cand}(k_2), A\ v, \mathcal{I}\ v)$

Let $v \in \mathsf{CVal}^{\mathsf{L}}(B_1)$ be given. By the definition of WT, it's enough to show that $\mathsf{WT}^{\theta}([v/x](\mathsf{Cand}(k_2)), A'\ v, \mathcal{I}\ v)$.

Now, $\mathsf{sizek}([v/x]k_2) < \mathsf{sizek}((x : B_1) \to k_2)$ and $A\ v \Rrightarrow A'\ v$, so the desired result will follow by induction if we can show that $\cdot \vdash A\ v : [v/x]k_2$. By TAppDep, it's enough to show that $\Gamma \vdash [v/x]k_2$. This follows by regularity (Lemma 5.2.25), inversion for dependent arrow kinds (Lemma 5.2.43) and substitution (Lemma 5.2.30).

The $\leftarrow$ case is similar.

- Suppose k is $(x : k_1) \to k_2$, so $\mathsf{Cand}(k)$ is $\mathsf{UKArr}\ (x : k_1)\ \mathsf{Cand}(k_2)$.

  We consider only the $\to$ direction—the other direction is similar. If $\mathcal{I} = \emptyset$, the result is immediate, so suppose instead that:

  – $\mathcal{I} : (\mathsf{TYP} \times [\mathsf{Cand}(k)]^{(\theta \times \mathbb{N})}) \to [U]$

  – and
$$\begin{aligned}&\forall B \in \mathsf{CTyp}(k_1), \forall \mathcal{I}' \in [\mathsf{Cand}(k_1)]^{(\theta\times\mathbb{N})},\\ &\quad B \Rrightarrow^* V\\ &\to \forall j, \forall \theta', \mathsf{WT}^{\theta'}(\mathsf{Cand}(k_1), B, \mathcal{I}'(\theta', j))\\ &\to \mathsf{WT}^{\theta}([B/x]\mathsf{Cand}(k_2), A\ B, \mathcal{I}(B, \mathcal{I}'))\end{aligned}$$

  Now let some $B \in \mathsf{CTyp}(k_1)$ and $\mathcal{I}' \in [\mathsf{Cand}(k_1)]^{(\theta\times\mathbb{N})}$ be given such that $B \Rrightarrow^* V$, and for any j and θ', $\mathsf{WT}^{\theta'}(\mathsf{Cand}(k), B, \mathcal{I}'(\theta', j))$.

  We must show that $\mathsf{WT}^{\theta}([B/x]\mathsf{Cand}(k_2), A'\ B, \mathcal{I}(B, \mathcal{I}'))$. But we know $\mathsf{WT}^{\theta}([B/x]\mathsf{Cand}(k_2), A\ B, \mathcal{I}(B, \mathcal{I}'))$ and $A\ B \Rrightarrow A'\ B$, so the result will follow by IH if we can show that $\cdot \vdash A\ B : [B/x]k_2$. This follows by regularity (Lemma 5.2.25), inversion for arrow kinds (Lemma 5.2.44) and substitution (Lemma 5.2.32).

□

Lemma 5.5.14. If $\cdot \vdash A : k$ and $A \Rrightarrow^* A'$ then $\mathsf{WT}^{\theta}(\mathsf{Cand}(k), A, \mathcal{I}) \leftrightarrow \mathsf{WT}^{\theta}(\mathsf{Cand}(k), A', \mathcal{I})$.

Proof. By induction on the proof of $A \Rrightarrow^* A'$, using Lemma 5.5.13. □

Lemma 5.5.15. Suppose $\cdot \vdash^{\mathsf{L}} b : B_1 = B_2$. If $\cdot \vdash [B_1/y]k$ and $\cdot \vdash [B_2/y]k$, then $\mathsf{WT}^{\theta}(\mathsf{Cand}([B_1/y]k), A, \mathcal{I}) \leftrightarrow \mathsf{WT}^{\theta}(\mathsf{Cand}([B_2/y]k), A, \mathcal{I})$.

Proof. By induction on $\mathsf{sizek}(k)$. If $k = s$, the result is immediate. In the remaining cases we show only the $\leftarrow$ direction. Since equality is symmetric (Lemma 5.2.36), the other direction follows immediately.

- Suppose k is $(x : A_1) \to k_2$.

  So we may assume that $\mathsf{WT}^\theta(\mathsf{UTArr}\ (x : [B_2/y]A_1)\ (\mathsf{Cand}([B_2/y]k_2)), A, \mathcal{I})$. If $\mathcal{I} = \emptyset$, the result is immediate, so suppose instead that

  - $\mathcal{I} : (\mathsf{VAL} \to [\mathsf{Cand}([B_2/y]k_2)])$, and
  - $\forall v \in \mathsf{CVal}^{\mathsf{L}}([B_2/y]A_1)$, $\mathsf{WT}^\theta([v/x]\mathsf{Cand}([B_2/y]k_2), A\ v, \mathcal{I}\ v)$.

  Observe that $[\mathsf{Cand}([B_2/y]k)] = [\mathsf{Cand}([B_1/y]k)]$ by Lemma 5.4.2, so $\mathcal{I} : \mathsf{VAL} \to [\mathsf{Cand}([B_1/y]k_2)]$ as well. Now let $v \in \mathsf{CVal}^{\mathsf{L}}([B_1/y]A_1)$ be given. We must show $\mathsf{WT}^\theta([v/x]\mathsf{Cand}([B_1/y]k_2), A\ v, \mathcal{I}\ v)$.

  But using Lemma 5.4.4 and since v, B_1 and B_2 are closed, we have $[v/x]\mathsf{Cand}([B_1/y]k_2) = \mathsf{Cand}([B_1/y][v/x]k_2)$, and similarly for $B2$. So the IH for $[v/x]k_2$ will yield the desired result if we can show that $v \in CValL([B2/y]A1)$. But this follows directly from EConvT, concluding the case.

- The case where k is $(x : k_1) \to k_2$ is similar.

$\square$

We are now almost ready to prove that the interpretation always returns a set in the appropriate WT relation, but first we must define a judgement that lifts WT over the context:

$$\frac{}{\mathsf{EnvWT}(\emptyset, \cdot)}\ \textsc{EWTNil} \qquad \frac{\begin{array}{c}\mathsf{EnvWT}(\rho, \Gamma) \\ \cdot \vdash^{\mathsf{L}} v : \rho\, A\end{array}}{\mathsf{EnvWT}(\rho[x \mapsto v], (\Gamma, x : A))}\ \textsc{EWTConsT}$$

$$\frac{\begin{array}{l}\mathsf{EnvWT}(\rho, \Gamma) \\ \cdot \vdash A : \rho\, k \qquad A \Rrightarrow^* V \\ \forall j, \forall \theta, \mathsf{WT}^\theta(\mathsf{Cand}(\rho\, k), A, \mathcal{I}(\theta, j))\end{array}}{\mathsf{EnvWT}(\rho[x \mapsto (A, \mathcal{I})], (\Gamma, x : k))}\ \textsc{EWTConsK}$$

Lemma 5.5.16. If $\mathsf{EnvWT}(\rho, \Gamma)$ then $\mathsf{EnvCand}(\rho, \Gamma)$.

Proof. By induction on the proof of $\mathsf{EnvWT}(\rho, \Gamma)$, using Lemma 5.5.12 in the EWT-ConsK case. $\square$

We will use the above fact implicitly in the lemma below, whenever $\mathsf{EnvCand}(\rho, \Gamma)$ is a prerequisite to another lemma.

Lemma 5.5.17 (The interpretation is well typed)**.** If $\Gamma \vdash A : k$ and $\mathsf{EnvWT}(\rho, \Gamma)$, then $\mathsf{WT}^\theta(\mathsf{Cand}(\rho\, k), \rho\, A, \mathcal{V}^\theta_\pi[\![A]\!]^j_\rho)$.

Proof. By induction on the derivation $\mathcal{E} :: \Gamma \vdash A : k$.

- $\mathcal{E} = \dfrac{(x : k) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : k}$ TVAR

  Since $\mathsf{EnvWT}(\rho, \Gamma)$, we know that $\rho = \rho_1[x \mapsto (B, \mathcal{I})]\, \rho_2$ for some ρ_1, ρ_2, B and $\mathcal{I}$ such that $\mathsf{WT}^{\theta}(\mathsf{Cand}(\rho_1\, k), B, \mathcal{I}(\theta, j))$.

  We must show that $\mathsf{WT}^{\theta}(\mathsf{Cand}(\rho\, k), \rho\, x, \mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho})$. By the definition of the interpretation, that is $\mathsf{WT}^{\theta}(\mathsf{Cand}(\rho\, k), B, \mathcal{I}(\theta, j))$. So it will be enough to show that $\rho_1\, k = \rho_1[x \mapsto (B, \mathcal{I})]\, \rho_2\, k$, which is true since $\vdash \Gamma$ by context regularity (Lemma 5.2.22).

- The cases for TARRCOMP, TARRPOLY, TNAT, TMU, TREFL, and TREFLT are immediate because $k = s$ and the interpretation is explicitly defined as a set of values of the appropriate type.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma, x : k_1 \vdash B : k_2} \qquad \Gamma \vdash (x : k_1) \to k_2}{\Gamma \vdash \lambda x : k_1.B : (x : k_1) \to k_2}$ TLAMTLC

  We must show that $\mathsf{WT}^{\theta}(\mathsf{Cand}((x : \rho\, k_1) \to \rho\, k_2), \lambda x : k_1.B, \mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1.B]\!]^{j}_{\rho})$. By definition:

  $$\mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1.B]\!]^{j}_{\rho} \qquad = (A \in \mathsf{TYP}, \mathcal{I} \in [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \mapsto \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto (A, \mathcal{I})]}$$

  By Lemmas 5.5.5 and 5.4.3, this function is either the empty set (in which case we are done) or has the type $(\mathsf{TYP} \times [\mathsf{Cand}(\rho\, k_1)]^{(\theta \times \mathbb{N})}) \to [\mathsf{Cand}(\rho\, k_2)]$.

  So let some $A \in \mathsf{CTyp}(\rho\, k_1)$ and $\mathcal{I} \in [\mathsf{Cand}(\rho\, k_1)]^{(\theta \times \mathbb{N})}$ be given such that $A \Rrightarrow^{*} V$ for some V, and $\mathsf{WT}^{\theta}(\mathsf{Cand}(\rho\, k_1), A, \mathcal{I}(\theta', j'))$ for any θ' and j'. By the definition of WT (and reducing the set-theoretic function above), it is enough to show that

  $$\mathsf{WT}^{\theta}([A/x]\mathsf{Cand}(\rho\, k_2), (\lambda x : \rho\, k_1.\rho\, B)\ A, \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto (A, \mathcal{I})]})$$

  Now, by EWTCONSK, $\mathsf{EnvWT}(\rho[x \mapsto (A, \mathcal{I})], (\Gamma, x : k_1))$. So the IH for $\mathcal{E}'$ yields that

  $$\mathsf{WT}^{\theta}(\mathsf{Cand}(\rho\,[A/x]k_2), \rho\,[A/x]B, \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto (A, \mathcal{I})]})$$

  But $\mathsf{Cand}(\rho\,[A/x]k_2) = [A/x]\mathsf{Cand}(\rho\, k_2)$ by Lemma 5.4.5. But $\rho\,[A/x]B \Rrightarrow^{*} \rho\,[V/x]B$ and $(\lambda x : \rho\, k_1.\rho\, B)\ A \Rrightarrow^{*} \rho\,[V/x]B$, so the desired result will follow from two uses of Lemma 5.5.14 if we can show that $\cdot \vdash \rho\,[A/x]B : \rho\,[A/x]k_2$ and $\cdot \vdash (\lambda x : \rho\, k_1.\rho\, B)\ A : \rho\,[A/x]k_2$. The former is immediate by Lemma 5.5.7, and the latter follows by TAPPTLC and Lemma 5.5.7.

- Case TLAMDEP is similar, but simpler.

- $$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \Gamma \vdash [A/x]k_2}{\Gamma \vdash B\ A : [A/x]k_2}\ \textsc{TAppTLC}$$

  We must show that $\mathsf{WT}^\theta(\mathsf{Cand}(\rho\,[A/x]k_2), \rho\,(B\ A), \mathcal{V}^\theta_\pi[\![B\ A]\!]^j_\rho)$.

  Now, if $\mathcal{V}^\theta_\pi[\![B\ A]\!]^j_\rho = \emptyset$, the result is immediate because $\mathsf{WT}^\theta(U, A', \emptyset)$ for any U and A'. So we may assume the other case of the interpretation of type-level applications applies, which means that $\rho\,A \rightsquigarrow^* V$ for some V and $(\rho\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_\rho)$ is in the domain of $\mathcal{V}^\theta_\pi[\![B]\!]^j_\rho$. We have:

  $$\mathcal{V}^\theta_\pi[\![B\ A]\!]^j_\rho = \mathcal{V}^\theta_\pi[\![B]\!]^j_\rho(\rho\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_\rho)$$

  The IH for $\mathcal{E}_1$ gives us that $\mathsf{WT}^\theta(\mathsf{Cand}(\rho\,((x : k_1) \to k_2)), \rho\,B, \mathcal{V}^\theta_\pi[\![B]\!]^j_\rho)$. Since we know the interpretation of B is non-empty (it's a function with a non-empty domain), this IH expands to:

  - $\mathcal{V}^\theta_\pi[\![B]\!]^j_\rho : (\mathsf{TYP} \times [\mathsf{Cand}(\rho\,k_1)]^{(\theta\times\mathbb{N})}) \to [\mathsf{Cand}(\rho\,k_2)]$
  - and:
  $$\begin{array}{l} \forall A' \in \mathsf{CTyp}(\rho\,k_1), \forall \mathcal{I} \in [\mathsf{Cand}(\rho\,k_1)]^{(\theta\times\mathbb{N})}, \\ \quad A' \Rrightarrow^* V \\ \to \forall j, \forall \theta', \mathsf{WT}^{\theta'}(\mathsf{Cand}(\rho\,k_1), A', \mathcal{I}'(\theta', j)) \\ \to \mathsf{WT}^\theta([A'/x]\mathsf{Cand}(\rho\,k_2), (\rho\,B)\ A', \mathcal{V}^\theta_\pi[\![B]\!]^j_\rho(A', \mathcal{I})) \end{array}$$

  But observe that $\rho\,A$ and $\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_\rho$ satisfy the requirements for A' and $\mathcal{I}$ here, since $\rho\,A$ parallel reduces to a value, and using the IH for $\mathcal{E}_2$. Thus, we may instantiate $\mathcal{E}_1$'s IH to obtain:

  $$\mathsf{WT}^\theta([\rho\,A/x]\mathsf{Cand}(\rho\,k_2), \rho\,(B\ A), \mathcal{V}^\theta_\pi[\![B]\!]^j_\rho(\rho\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_\rho))$$

  And since $[\rho\,A/x]\mathsf{Cand}(\rho\,k_2) = \mathsf{Cand}(\rho\,[A/x]k_2)$ by Lemma 5.4.5, this concludes the case.

- Case TAppDep is similar, but simpler.

- Cases TMonoPoly, TAt, TSum and TSigma are straightforward by induction.

- $$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash^{\mathsf{L}} b : B_1 = B_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : [B_1/x]k} \qquad \Gamma \vdash [B_2/x]k}{\Gamma \vdash A : [B_2/x]k}\ \textsc{TConvT}$$

  In this case, we must show that $\mathsf{WT}^\theta(\mathsf{Cand}(\rho\,[B_2/x]k), \rho\,A, \mathcal{V}^\theta_\pi[\![A]\!]^j_\rho)$. The IH for $\mathcal{D}_2$ yields that $\mathsf{WT}^\theta(\mathsf{Cand}(\rho\,[B_1/x]k), \rho\,A, \mathcal{V}^\theta_\pi[\![A]\!]^j_\rho)$. Lemma 5.5.15 bridges the gap.

□

In practice, we only need the special case of this lemma where k is a sort s:

Lemma 5.5.18. Suppose $\Gamma \vdash A : s$ and $\mathsf{EnvWT}(\rho, \Gamma)$. Then $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho} \subseteq \mathsf{CVal}^{\theta}(\rho\, A)$.

Proof. By Lemma 5.5.17, unfolding the definition of WT. □

Another essential lemma that must be generalized due to the presence of type-level computation is downward closure. In previous chapters, we were able to state downward closure as, roughly, if $i \leq j$ then $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho} \subseteq \mathcal{V}^{\theta}_{\pi}[\![A]\!]^{i}_{\rho}$. The use of $\subseteq$ here is problematic in the case where A is a type-level function. The solution is to generalize $\subseteq$ as follows:

Definition 5.5.19. Define the property $\mathcal{I}_1 \sqsubseteq^{U} \mathcal{I}_2$ by recursion on the size of U, as follows:

- $\mathcal{I}_1 \sqsubseteq^{U} \mathcal{I}_2$ iff $\mathcal{I}_1, \mathcal{I}_2 \in \mathsf{VAL}$ and $\mathcal{I}_1 \subseteq \mathcal{I}_2$.
- $\mathcal{I}_1 \sqsubseteq^{\mathsf{UTArr}\ (x:A_1)\ U_2} \mathcal{I}_2$ iff $\mathcal{I}_1 = \mathcal{I}_2 = \emptyset$, or
  - $\mathcal{I}_1, \mathcal{I}_2 : (\mathsf{VAL} \to [U])$, and
  - $\forall v \in \mathsf{CVal}^{\mathsf{L}}(A_1)$, $\mathcal{I}_1 \sqsubseteq^{[v/x]\,U_2} \mathcal{I}_2$
- $\mathcal{I}_1 \sqsubseteq^{\mathsf{UKArr}\ (x:k_1)\ U_2} \mathcal{I}_2$ iff $\mathcal{I}_1 = \mathcal{I}_2 = \emptyset$, or
  - $\mathcal{I}_1, \mathcal{I}_2 : (\mathsf{TYP} \times [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \to [U_2]$
  - and:

$$\begin{aligned}
&\forall B \in \mathsf{CTyp}(k_1), \forall \mathcal{I}_1', \mathcal{I}_2' \in [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})},\\
&\quad B \Rrightarrow^{*} V\\
&\to \forall i_1 \leq i_2, \forall \theta', \mathcal{I}_1'(\theta', i_2) \sqsubseteq^{\mathsf{Cand}(k_1)} \mathcal{I}_2'(\theta', i_1)\\
&\to \mathcal{I}_1(B, \mathcal{I}_1') \sqsubseteq^{[B/x]\,U_2} \mathcal{I}_2(B, \mathcal{I}_2')
\end{aligned}$$

The following lemma about this definition will be useful in the TConvT case of the downward closure lemma:

Lemma 5.5.20. Suppose $\cdot \vdash^{\mathsf{L}} b : B_1 = B_2$. If $\cdot \vdash [B_1/y]k$ and $\cdot \vdash [B_2/y]k$, then $(\mathcal{I}_1 \sqsubseteq^{\mathsf{Cand}([B_1/y]k)} \mathcal{I}_2) \leftrightarrow (\mathcal{I}_1 \sqsubseteq^{\mathsf{Cand}([B_2/y]k)} \mathcal{I}_2)$.

Proof. By induction on $\mathsf{sizek}(k)$, similar to Lemma 5.5.15. □

Predictably, we need a judgement that lifts the relevant uses of this relation over an environment.

$$\frac{}{\emptyset \sqsubseteq^{\cdot}_{\mathsf{DC}} \emptyset}\ \textsc{EDCNil} \qquad \frac{\begin{array}{c}\rho_1 \sqsubseteq^{\Gamma}_{\mathsf{DC}} \rho_2 \\ \cdot \vdash^{\mathsf{L}} v : \rho_1\, A\end{array}}{\rho_1[x \mapsto v] \sqsubseteq^{(\Gamma, x:A)}_{\mathsf{DC}} \rho_2[x \mapsto v]}\ \textsc{EDCConsT}$$

$$\frac{\begin{array}{l}\rho_1 \sqsubseteq^{\Gamma}_{\mathsf{DC}} \rho_2 \\ \cdot \vdash A : \rho_1\, k \qquad A \Rrightarrow^{*} V \\ \forall i_1 \leq i_2, \forall \theta, \mathcal{I}_1(\theta, i_2) \sqsubseteq^{\mathsf{Cand}(\rho\, k)} \mathcal{I}_2(\theta, i_1)\end{array}}{\rho_1[x \mapsto (A, \mathcal{I}_1)] \sqsubseteq^{\Gamma}_{\mathsf{DC}} \rho_2[x \mapsto (A, \mathcal{I}_2)]}\ \textsc{EDCConsK}$$

Lemma 5.5.21 (Downward closure)**.** Suppose $\Gamma \vdash A : k$ and $\rho_1 \sqsubseteq^{\Gamma}_{\mathsf{DC}} \rho_2$. If $j_1 \leq j_2$, then $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j_2}_{\rho_1} \sqsubseteq^{\mathsf{Cand}(\rho\, k)} \mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j_1}_{\rho_2}$.

Proof. By induction on the derivation $\mathcal{E} :: \Gamma \vdash A : k$. Most cases are straightforward by the definition of the interpretation and induction. We show two cases in more detail:

- $$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \Gamma \vdash [A/x]k_2}{\Gamma \vdash B\ A : [A/x]k_2}\ \textsc{TAppTLC}$$

  We must show that $\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j_2}_{\rho_1} \sqsubseteq^{\mathsf{Cand}(\rho_1\, [A/x]k_2)} \mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j_1}_{\rho_2}$. By the IH for $\mathcal{E}_1$, we know that either the interpretation of B at both j_1 and j_2 is the empty set, in which case we are done, or that in both cases it has the type $(\mathsf{TYP} \times [\mathsf{Cand}(\rho\, k_1)]^{(\theta \times \mathbb{N})}) \to [\mathsf{Cand}(\rho\, k_2)]$ and:

$$\begin{array}{l}\forall A' \in \mathsf{CTyp}(\rho\, k_1), \forall \mathcal{I}'_1, \mathcal{I}'_2 \in [\mathsf{Cand}(\rho\, k_1)]^{(\theta \times \mathbb{N})}, \\ \quad A' \Rrightarrow^{*} V \\ \to \forall i_1 \leq i_2, \forall \theta', \mathcal{I}'_1(\theta', i_2) \sqsubseteq^{\mathsf{Cand}(k_1)} \mathcal{I}'_2(\theta', i_1) \\ \to \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j_2}_{\rho_1}(A', \mathcal{I}'_1) \sqsubseteq^{[A'/x]\mathsf{Cand}(\rho\, k_2)} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j_1}_{\rho_1}(A', \mathcal{I}'_2)\end{array}$$

  We'd like to show that $\rho\, A$, $\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j_2]}_{\rho}$ and $\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j_1]}_{\rho}$ satisfy the requirements of $A', \mathcal{I}'_1$ and $\mathcal{I}'_2$ above. We may assume that $\rho\, A \rightsquigarrow^{*} V$ for some V, since otherwise the interpretation of BA is always the empty set, in which case we are done. The requirements on $\mathcal{I}'_1$ and $\mathcal{I}'_2$ mean we must show that:

$$\forall i_1 \leq i_2, \forall \theta', \mathcal{V}^{\theta'}_{\tau}[\![A]\!]^{\mathsf{min}(j_2, i_2)}_{\rho_1} \sqsubseteq^{\mathsf{Cand}(\rho_1\, k_1)} \mathcal{V}^{\theta'}_{\tau}[\![A]\!]^{\mathsf{min}(j_1, i_1)}_{\rho_2}$$

  This will follow by the IH for $\mathcal{E}_2$ if we can show that $\mathsf{min}(j_1, i_1) \leq \mathsf{min}(j_2, i_2)$. Since we know $i_1 \leq i_2$ and $j_1 \leq j_2$, this holds by a simple arithmetic case analysis.

  So, instantiating the $\mathcal{E}_1$ IH above, we find that:

$$\mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j_2}_{\rho_1}(A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j_2]}_{\rho_1}) \sqsubseteq^{[\rho\, A/x]\mathsf{Cand}(\rho_1\, k_2)} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j_1}_{\rho_1}(A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j_1]}_{\rho_2})$$

But $[\rho\, A/x]\mathsf{Cand}(\rho_1\, k_2) = \mathsf{Cand}(\rho_1\, [A/x]k_2)$, so it will be enough to show that:

$$\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j_2}_{\rho_1} = \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j_2}_{\rho_1}(A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j_2]}_{\rho_1})$$

and

$$\mathcal{V}^{\theta}_{\pi}[\![B\ A]\!]^{j_1}_{\rho_2} = \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j_1}_{\rho_1}(A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j_1]}_{\rho_2})$$

This follows the definition of the interpretation, the observation we have already made about the type of B's interpretation, and Lemma 5.5.5.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash^{\mathsf{L}} b : B_1 = B_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : [B_1/x]k} \qquad \Gamma \vdash [B_2/x]k}{\Gamma \vdash A : [B_2/x]k}$ TConvT

  By induction and Lemma 5.5.20.

□

In practice, we only need the special case of this lemma where k is a sort s and there is only one environment:

Lemma 5.5.22 (Downward closure for proper types)**.** Suppose $\Gamma \vdash A : s$ and $\rho \sqsubseteq^{\Gamma}_{\mathsf{DC}} \rho$. If $j_1 \leq j_2$, then $\Omega^{\theta}_{\pi}[\![A]\!]^{j_2}_{\rho} \subseteq \Omega^{\theta}_{\pi}[\![A]\!]^{j_1}_{\rho}$.

Proof. By the definition of the interpretation and Lemma 5.5.21, unfolding the definition of $\sqsubseteq^{U}$. □

Lemma 5.5.23 (Downward closure for kinds)**.** For any kind k and environment ρ, if $j_1 \leq j_2$ then $\Omega_{\pi}[\![k]\!]^{j_2}_{\rho} \subseteq \Omega_{\pi}[\![k]\!]^{j_1}_{\rho}$.

Proof. By examining the definition of the interpretation for each of the possible forms of k. □

Another similar lemma about previous systems said that the logical interpretation of a type is a subset of its programmatic interpretation, modeling the idea that logical terms may always be used in the programmatic fragment, but not vice versa. We will reuse the $\sqsubseteq$ relation to state this lemma for PCC$^{\theta}$. Following the pattern established thus far, we first define a judgement that lifts this property over environments:

$$\frac{}{\emptyset \sqsubseteq_{\mathsf{LP}} \emptyset}\ \text{ELPNil} \qquad \frac{\begin{array}{c}\rho_1 \sqsubseteq^{\Gamma}_{\mathsf{LP}} \rho_2 \\ \cdot \vdash^{\mathsf{L}} v : \rho_1\, A\end{array}}{\rho_1[x \mapsto v] \sqsubseteq^{(\Gamma,x:A)}_{\mathsf{LP}} \rho_2[x \mapsto v]}\ \text{ELPConsT}$$

$$\frac{\begin{array}{l}\rho_1 \sqsubseteq^{\Gamma}_{\mathsf{LP}} \rho_2 \\ \cdot \vdash A : \rho_1\, k \quad A \Rrightarrow^{*} V \\ \forall i, \mathcal{I}_1(\mathsf{L}, i) \sqsubseteq^{\mathsf{Cand}(\rho\, k)} \mathcal{I}_2(\mathsf{P}, i)\end{array}}{\rho_1[x \mapsto (A, \mathcal{I}_1)] \sqsubseteq^{\Gamma}_{\mathsf{LP}} \rho_2[x \mapsto (A, \mathcal{I}_2)]}\ \text{ELPConsK}$$

Lemma 5.5.24 (The value interpretation models subsumption)**.** Suppose $\Gamma \vdash A : k$ and $\rho_1 \sqsubseteq^{\Gamma}_{\mathsf{LP}} \rho_2$. Then, for any π and j, $\mathcal{V}^{\mathsf{L}}_{\pi}[\![A]\!]^{j}_{\rho_1} \sqsubseteq^{\mathsf{Cand}(\rho_1\, k)} \mathcal{V}^{\mathsf{P}}_{\pi}[\![A]\!]^{j}_{\rho_2}$.

Proof. Let $\mathcal{E} :: \Gamma \vdash A : k$ be given. We proceed by induction on the lexicographically ordered pair $(\pi, \mathcal{E})$. Most cases are straightforward by induction. We show one in more detail:

- $$\mathcal{D} = \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash k} \qquad \overset{\mathcal{E}_2}{\Gamma, x : k \vdash B : \star_\sigma}}{\Gamma \vdash (x : k) \to B : \star_\sigma}\ \textsc{TArrPoly}$$

  Here, according to the definition of $\sqsubseteq^{\mathsf{UBase}}$, we must show that $\mathcal{V}^{\mathsf{L}}_{\pi}[\![(x : k) \to B]\!]^{j}_{\rho_1} \subseteq \mathcal{V}^{\mathsf{P}}_{\pi}[\![(x : k) \to B]\!]^{j}_{\rho_2}$. If π is τ, then both interpretations are the empty set and the case is trivial. So, suppose $\pi = \sigma$. Unfolding the definition of the interpretation, we see it is sufficient to show that:

  $$\begin{aligned} &\{\lambda x{:}k'.b \mid && \cdot \vdash^{\mathsf{L}} \lambda x{:}k'.b : \rho_1\,((x : k) \to B) \\ & && \text{and } \forall i \leq j,\, \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho_1}, \\ & && \quad [V/x]b \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho_1[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}\} \\ \subseteq\ &\{\lambda x{:}k'.b \mid && \cdot \vdash^{\mathsf{P}} \lambda x{:}k'.b : \rho_2\,((x : k) \to B) \\ & && \text{and } \forall i < j,\, \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho_2}, \\ & && \quad [V/x]b \in \mathcal{C}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho_2[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]} \end{aligned}$$

  So let some $\lambda x : k'.b$ in the former set be given. Since the two environments share the same term and type bindings, $\rho_1\,((x : k) \to B) = \rho_2\,((x : k) \to B)$, and thus $\lambda x{:}k'.bn$ satisfies the typing requirement of the second set by TSub.

  So, let some $i \leq j$ and $V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho}$ be given. Since $\lambda x{:}k'.b$ is in the former set, we know that $[V/x]b \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}$. Unfolding the definition of the computational interpretation, we have:

  $$[V/x]b \rightsquigarrow^{*} v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}$$

  We must show that, if $[V/x]b \rightsquigarrow^{i'} v$ for some $i' \leq i$, then:

  $$v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{i'}_{\rho[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}$$

  This will follow from downward closure (Lemma 5.5.22) if we can show:

  $$v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}$$

  This in turn will follow from the IH for $(\mathsf{up}(B), \mathcal{E}_2)$, if we can show that:

  $$\rho_1[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})] \sqsubseteq^{\Gamma, x:k}_{\mathsf{LP}} \rho_2[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]$$

This follows from ELPConsK if we can show that, for any $i' \leq i$:

$$\mathcal{V}^{\mathsf{L}}_{\tau}[\![V]\!]^{i'}_{\emptyset} \sqsubseteq^{\mathsf{Cand}(\rho_1\, k)} \mathcal{V}^{\mathsf{P}}_{\tau}[\![V]\!]^{i'}_{\emptyset}$$

But we know by Lemma 5.5.18 that there is some derivation $\mathcal{E}' :: \cdot \vdash V : \rho_1\, k$. So, this is exactly the IH for $(\tau, \mathcal{E}')$.

□

Lemma 5.5.25 (The computational interpretation models subsumption)**.** Suppose $\Gamma \vdash A : s$ and $\rho \sqsubseteq^{\Gamma}_{\mathsf{LP}} \rho$. Then, for any π and j, $\mathcal{C}^{\mathsf{L}}_{\pi}[\![A]\!]^{j}_{\rho} \subseteq \mathcal{C}^{\mathsf{P}}_{\pi}[\![A]\!]^{j}_{\rho}$.

Proof. By Lemma 5.5.24 and the definition of the interpretation. □

5.6 A Notion of Equivalence for Interpretations

Simple set-theoretic equality is often insufficient for reasoning about the equivalence of interpretations, for reasons similar to those that necessitated "universe-indexed" versions of lemmas in the previous section. We have interpreted type-level functions as set theoretic functions with very large domains:

$$\mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1 . B]\!]^{j}_{\rho} \qquad = (A \in \mathsf{TYP}, \mathcal{I} \in [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \mapsto \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto (A,\mathcal{I})]}$$

This was convenient for defining the interpretation, but it is inconvenient for proving facts about the interpretation. The issue is that this function's domain is much larger than the actual class of arguments for which we care about its behavior. First, we only care about types A that appear in the interpretation of $\rho\, k_1$, but the domain allows all closed types. Second, this function will only be called in the very specific case that $\mathcal{I}$ is the interpretation of A (i.e., $\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\pi}[\![A]\!]^{[0..j]}_{\rho}$), but the domain allows any set-theoretic object $\mathcal{I}$ from the universe of possible interpretations of types of kind k_1.

The solution is to define a better notion of equivalence which only demands that the set-theoretic functions agree when applied to sensible arguments. We write $\mathcal{I}_1 \cong^{U}_{j} \mathcal{I}_2$ to indicate that $\mathcal{I}_1$ and $\mathcal{I}_2$ are equivalent set-theoretic objects such that $\mathcal{I}_1, \mathcal{I}_2 \in [U]$.

Definition 5.6.1 (Equivalence of interpretations)**.** We define the relation $\mathcal{I}_1 \cong^{U}_{j} \mathcal{I}_2$ recursively by the following three cases:

- $\mathcal{I} \cong^{\mathsf{UBase}}_{j} \mathcal{I}$ iff $\mathcal{I} \in \mathcal{P}(\mathsf{VAL})$.
- $\mathcal{I}_1 \cong^{\mathsf{UTArr}\ (x:A)\ U_2}_{j} \mathcal{I}_2$ iff $\mathcal{I}_1 = \mathcal{I}_2 = \emptyset$ or
  - $\mathcal{I}_1, \mathcal{I}_2 : \mathsf{VAL} \to [U_2]$,

- and

$$\begin{aligned}&\forall v_1, v_2 \in \mathsf{CVal}^{\mathsf{L}}(A)\\ &\quad v_1 \Rrightarrow^* v' \wedge v_2 \Rrightarrow^* v'\\ &\rightarrow \mathcal{I}_1\; v_1 \cong_j^{[v'/x]\,U_2} \mathcal{I}_2\; v_2\end{aligned}$$

- $\mathcal{I}_1 \cong_j^{\mathsf{UKArr}\ (x:k_1)\ U_2} \mathcal{I}_2$ iff $\mathcal{I}_1 = \mathcal{I}_2 = \emptyset$ or
  - $\mathcal{I}_1, \mathcal{I}_2 : (\mathsf{TYP} \times [\mathsf{Cand}(k_1)]^{(\theta \times \mathbb{N})}) \rightarrow [U_2]$
  - and,

$$\begin{aligned}&\forall B_1, B_2 \in \mathsf{CTyp}(k_1), \forall \mathcal{I}_1', \mathcal{I}_2' \in [\mathsf{Cand}(k_1)]^{(\theta\times\mathbb{N})}\\ &\quad B_1 \Rrightarrow^* V \wedge B_2 \Rrightarrow^* V\\ &\rightarrow \forall i \leq j, \forall \theta, \mathcal{I}_1'(\theta, i) \cong_i^{\mathsf{Cand}(k_1)} \mathcal{V}_\tau^\theta[\![B_1]\!]_\emptyset^i\\ &\rightarrow \forall i \leq j, \forall \theta, \mathcal{I}_2'(\theta, i) \cong_i^{\mathsf{Cand}(k_1)} \mathcal{V}_\tau^\theta[\![B_2]\!]_\emptyset^i\\ &\rightarrow \forall i \leq j, \forall \theta, \mathcal{I}_1'(\theta, i) \cong_i^{\mathsf{Cand}(k_1)} \mathcal{I}_2'(\theta, i)\\ &\rightarrow \forall i \leq j, \mathcal{I}_1(B_1, \mathcal{I}_1') \cong_i^{[V/x]\,U_2} \mathcal{I}_2(B_2, \mathcal{I}_2')\end{aligned}$$

Like our previous universe-indexed definitions, we can see that this property is well defined by recursion on $\mathsf{sizeU}(U)$.

While we refer to $\cong_k^U$ as an "equivalence relation", it is not actually reflexive for every $\mathcal{I} \in [U]$. In particular, for functions $\mathcal{I}$ in $[\mathsf{UTArr}\ (x : A_1)\ U_2]$ or $[\mathsf{UKArr}\ (x : k_1)\ U_2]$, the relation is only reflexive when I does not distinguish between inputs that are equivalent. When I is the output of the interpretation, this is always the case. However, it will take a bit more effort to prove this formally.

In the meantime, we can at least observe that $\cong_j^U$ is symmetric and transitive. These two lemmas may be proved directly by induction on $\mathsf{sizeU}(U)$, examining the definition of the relation in each of the three cases.

Lemma 5.6.2 ($\cong$ is symmetric)**.** For any U and j, if $\mathcal{I}_1 \cong_j^U \mathcal{I}_2$, then $\mathcal{I}_2 \cong_j^U \mathcal{I}_1$.

Lemma 5.6.3 ($\cong$ is transitive)**.** For any U and j, if $\mathcal{I}_1 \cong_j^U \mathcal{I}_2$ and $\mathcal{I}_2 \cong_j^U \mathcal{I}_3$, then $\mathcal{I}_1 \cong_j^U \mathcal{I}_3$.

We may also prove that this equivalence respects provable type and term equalities. Both of these lemmas are proved by induction on $\mathsf{sizeU}(U)$, examining the definition of the equivalence for each possible form of U.

Lemma 5.6.4. If $\cdot \vdash^{\mathsf{L}} a : b_1 = b_2$ then $(\mathcal{I}_1 \cong_j^{[b_1/x]\,U} \mathcal{I}_2) \leftrightarrow (\mathcal{I}_1 \cong_j^{[b_2/x]\,U} \mathcal{I}_2)$.

Lemma 5.6.5. If $\cdot \vdash^{\mathsf{L}} a : B_1 = B_2$ then $(\mathcal{I}_1 \cong_j^{[B_1/x]\,U} \mathcal{I}_2) \leftrightarrow (\mathcal{I}_1 \cong_j^{[B_2/x]\,U} \mathcal{I}_2)$.

To prove many of the lemmas about our new equivalence, we must extend it to environments. Roughly speaking, two environments are equivalent just if the interpretations contained in them are:

$$\frac{}{\emptyset \cong_j^{\cdot} \emptyset}\ \textsc{EEqNil} \qquad \frac{\begin{array}{c}\rho_1 \cong_j^{\Gamma} \rho_2 \\ v_1 \Rrightarrow^* v \qquad v_2 \Rrightarrow^* v \\ \cdot \vdash^{\mathsf{L}} v_1 : \rho_1\, A \qquad \cdot \vdash^{\mathsf{L}} v_2 : \rho_2\, A\end{array}}{\rho_1[x \mapsto v_1] \cong_j^{\Gamma, x:A} \rho_2[x \mapsto v_2]}\ \textsc{EEqConsT}$$

$$\frac{\begin{array}{c}\rho_1 \cong_j^{\Gamma} \rho_2 \\ B_1 \Rrightarrow^* V \qquad B_2 \Rrightarrow^* V \\ \cdot \vdash B_1 : \rho_1\, k \qquad \cdot \vdash B_2 : \rho_2\, k \\ \forall i \leq j, \forall \theta, \mathcal{I}_1(\theta, i) \cong_i^{\mathsf{Cand}(\rho_1\, k)} \mathcal{V}_\tau^\theta[\![B_1]\!]_\emptyset^j \\ \forall i \leq j, \forall \theta, \mathcal{I}_2(\theta, i) \cong_i^{\mathsf{Cand}(\rho_1\, k)} \mathcal{V}_\tau^\theta[\![B_2]\!]_\emptyset^j \\ \forall i \leq j, \forall \theta, \mathcal{I}_1(\theta, i) \cong_i^{\mathsf{Cand}(\rho_1\, k)} \mathcal{I}_2(\theta, i)\end{array}}{\rho_1[x \mapsto (B_1, \mathcal{I}_1)] \cong_j^{\Gamma, x:k} \rho_2[x \mapsto (B_2, \mathcal{I}_2)]}\ \textsc{EEqConsK}$$

Unsurprisingly, we need a few lemmas about this relation.

Lemma 5.6.6 ($\cong_j^\Gamma$ is downward closed)**.** If $\rho_1 \cong_j^\Gamma \rho_2$ and $i \leq j$ then $\rho_1 \cong_i^\Gamma \rho_2$.

Proof. By induction on the proof of $\rho_1 \cong_j^\Gamma \rho_2$. ☐

Lemma 5.6.7. If $\Gamma \vdash A : s$ and $\rho_1 \cong_j^\Gamma \rho_2$, then $\cdot \vdash^\theta a : \rho_1\, A$ iff $\cdot \vdash^\theta a : \rho_2\, A$.

Proof. The value and type bindings in ρ_1 and ρ_2 are assumed to be well typed and corresponding value or type bindings are provably equal, since they reduce to the same value. Thus, this lemma may be proved in either direction by inducting on the typing derivation and using TConv or TConvT when a variable is encountered (though it must first be generalized to handle open contexts). ☐

Lemma 5.6.8. If $\vdash \Gamma$ and $\rho_1 \cong_j^\Gamma \rho_2$ then $\mathsf{EnvTyp}(\rho_1, \Gamma)$ and $\mathsf{EnvTyp}(\rho_2, \Gamma)$.

Proof. By induction on the proof of $\rho_1 \cong_j^\Gamma \rho_2$. ☐

5.7 Main Interpretation Lemmas

In this section, we will prove the two most complicated and important lemmas about the interpretation (along with several corollaries). The first is that bindings from the environment ρ may be substituted into the type or kind being interpreted without changing its interpretation. The second is that reduction does not change the interpretation of a type or kind.

These lemmas are difficult to prove, for several reasons. First, we must work with the complicated notion of equivalence defined above. Second, the lemmas must be proved by induction over metrics similar to the one we used to prove the interpretation was well founded. To make this inductive structure clear, we state both lemmas in a somewhat stilted way, being careful to quantify the elements of the relevant quadruple at the front.

While these results are quite involved, most of the challenging theoretical work was in setting up our definitions appropriately. The proofs are primarily exercises in checking that our definitions line up appropriately and identifying a few key lemmas.

Lemma 5.7.1 (Environment Shifting)**.** Let some π and j be given. Suppose we have a proof $\mathcal{E} :: \Gamma \vdash A : k$ or a proof $\mathcal{F} :: \Gamma \vdash k$, and let some Ω be given. If $\rho_1\,\rho_2\,\rho_3 \cong^{\Gamma}_{j} \rho_1\,\rho_2\,\rho'_3$, then:

- If we have a proof $\mathcal{E} :: \Gamma \vdash A : k$ and $\pi = \mathsf{up}(A)$, then:
  - If $\Omega = \mathcal{V}$, then $\mathcal{V}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)}_{j} \mathcal{V}^{\theta}_{\pi}[\![\rho_2\,A]\!]^{j}_{\rho_1\,\rho'_3}$.
  - If $\Omega = \mathcal{C}$ and $k = s$, then $\mathcal{C}^{\theta}_{\pi}[\![A]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} = \mathcal{C}^{\theta}_{\pi}[\![\rho_2\,A]\!]^{j}_{\rho_1\,\rho_3}$.
- If we have a proof $\mathcal{F} :: \Gamma \vdash k$ and $\pi = \mathsf{up}(k)$, then

$$\Omega_{\mathsf{up}(k)}[\![k]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} = \Omega_{\mathsf{up}(\rho_2\,k)}[\![\rho_2\,k]\!]^{j}_{\rho_1\,\rho'_3}.$$

Proof. By lexicographic induction on the quadruple $(\pi, j, \mathcal{D}/\mathcal{E}, \Omega)$. It is sensible to consider $\mathcal{D}$ and $\mathcal{E}$ as one argument to the same induction since they are mutually defined, though it is equivalent to prove this theorem by four nested inductions where the third is by mutual induction on these derivations.

Note that, by Lemmas 5.5.8 and 5.6.8, $\mathsf{up}(A) = \mathsf{up}(\rho_2\,(A))$ in the cases where $\mathcal{E} :: \Gamma \vdash A : k$, and $\mathsf{up}(k) = \mathsf{up}(\rho_2\,(k))$ in the cases where $\mathcal{F} :: \Gamma \vdash k$. We will use these facts repeatedly and implicitly in the remainder of the proof, to avoid getting bogged down with dozens of references to these two lemmas.

We begin with the cases where $\Omega = \mathcal{C}$. There are three:

- Suppose $\mathcal{E} :: \Gamma \vdash A : s$ and $\Omega = \mathcal{C}$ and $\theta = \mathsf{L}$. We must show that:

$$\begin{aligned} & \{a \mid \cdot \vdash^{\mathsf{L}} a : \rho_1\,\rho_2\,\rho_3\,A \text{ and } a \rightsquigarrow^{*} v \in \mathcal{V}^{\mathsf{L}}_{\pi}[\![A]\!]^{j}_{\rho_1\,\rho_2\,\rho_3}\} \\ = \; & \{a \mid \cdot \vdash^{\mathsf{L}} a : \rho_1\,\rho'_3\,\rho_2\,A \text{ and } a \rightsquigarrow^{*} v \in \mathcal{V}^{\mathsf{L}}_{\pi}[\![\rho_2\,A]\!]^{j}_{\rho_1\,\rho'_3}\} \end{aligned}$$

  Now $\rho_1\,\rho_2\,\rho'_3\,A = \rho_1\,\rho'_3\,\rho_2\,A$, and, by 5.6.7 $\cdot \vdash^{\mathsf{L}} a : \rho_1\,\rho_2\,\rho_3\,A$ iff $\cdot \vdash^{\mathsf{L}} a : \rho_1\,\rho_2\,\rho'_3\,A$. So, it is enough to show that $\mathcal{V}^{\mathsf{L}}_{\pi}[\![A]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} = \mathcal{V}^{\mathsf{L}}_{\pi}[\![\rho_2\,A]\!]^{j}_{\rho_1\,\rho'_3}$. This is exactly the IH for $(\pi, j, \mathcal{E}, \mathcal{V})$.

- The case when $\mathcal{E} :: \Gamma \vdash A : s$ and $\Omega = \mathcal{C}$ and $\theta = \mathsf{P}$ is similar, except that the use of the IH also requires us to observe that $\cong^{\Gamma}_{j}$ is downward closed (Lemma 5.6.6).

- The case when $\mathcal{F} :: \Gamma \vdash k$ and $\Omega = \mathcal{C}$ is similar.

For the cases where $\Omega = \mathcal{V}$, we consider the possible derivations of $\mathcal{E}$ or $\mathcal{F}$.

- $\mathcal{E} = \dfrac{(x:k) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : k}$ TVar

  We must show that $\mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)}_{j} \mathcal{V}^{\theta}_{\pi}[\![\rho_2\,x]\!]^{j}_{\rho_1\,\rho'_3}$. Either $x \in \mathsf{dom}(\rho_2)$ or not:

  - Suppose $x \in \mathsf{dom}(\rho_2)$. So $\rho_2 = \rho_{21}[x \mapsto (B, \mathcal{I})]\,\rho_{22}$ for some B and $\mathcal{I}$.

    By the definition of the interpretation, it is enough to show that:

    $$\mathcal{I}(\theta, j) \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)}_{j} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho_1\,\rho'_3}$$

    Now, since $\rho_1\,\rho_{21}[x \mapsto (B,\mathcal{I})]\,\rho_{22}\,\rho_3 \cong^{\Gamma}_{j} \rho_1\,\rho_{21}[x \mapsto (B,\mathcal{I})]\,\rho_{22}\,\rho'_3$, we know that $\mathcal{I}(\theta, j) \cong^{\mathsf{Cand}(\rho_1\,\rho_{21}\,k)}_{j} \mathcal{V}^{\theta}_{\tau}[\![B]\!]^{j}_{\emptyset}$ and that $\cdot \vdash B : \rho_1\,\rho_{21}\,k$ by the definition of $\cong^{\Gamma}_{j}$. Since $\rho_1\,\rho_{21}\,k$ is closed, $\mathsf{Cand}(\rho_1\,\rho_{21}\,k) = \mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)$. So, to conclude the case it will be enough to show that:

    $$\mathcal{V}^{\theta}_{\tau}[\![B]\!]^{j}_{\emptyset} = \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho_1\,\rho'_3}$$

    But $\pi = \mathsf{up}(x) = \tau$. The result then follows by Lemma 5.5.10, since B is closed.

  - Suppose instead that $x \notin \mathsf{dom}(\rho_2)$.

    By Lemma 5.5.10, $\mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} = \mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho_3}$, so it will be enough to show that $\mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)}_{j} \mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho'_3}$. But, by the definition of the interpretation and since $\rho_1\,\rho_2\,\rho_3 \cong^{\Gamma}_{j} \rho_1\,\rho_2\,\rho'_3$, we know that $\mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho_3} = \mathcal{I}_1$ and $\mathcal{V}^{\theta}_{\pi}[\![x]\!]^{j}_{\rho_1\,\rho_3} = \mathcal{I}_2$ for some $\mathcal{I}_1$ and $\mathcal{I}_2$ such that $\mathcal{I}_1 \cong^{\mathsf{Cand}(\rho'\,k)}_{j} \mathcal{I}_2$ where ρ' is some initial prefix of $\rho_1\,\rho_2\,\rho_3$ that contains all the variables of k. This concludes the case, since therefore $\mathsf{Cand}(\rho'\,k) = \mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)$.

- $\mathcal{E} = \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash k} \qquad \overset{\mathcal{E}_2}{\Gamma, x : k \vdash B : \star_\sigma}}{\Gamma \vdash (x:k) \to B : \star_\sigma}$ TArrPoly

  Here $\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k) = \mathsf{Cand}(\star_\sigma) = \mathsf{UBase}$. Additionally, $\pi = \mathsf{up}((x:k) \to B) = \sigma$, so the IH will be available for any π'. We will consider only the case where $\theta = \mathsf{L}$, as the other is similar. Recall:

  $$\begin{aligned} \mathcal{V}^{\mathsf{L}}_{\sigma}[\![(x:k) \to B]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} \quad &= \\ \{\lambda x\!:\!k'.b \mid \quad &\cdot \vdash^{\mathsf{L}} \lambda x\!:\!k'.b : \rho_1\,\rho_2\,\rho_3\,((x:k) \to B) \\ &\text{and } \forall i \le j,\ \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho_1\,\rho_2\,\rho_3}, \\ &\quad [V/x]b \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho_1\,\rho_2\,\rho_3[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}\} \end{aligned}$$

  Now $\rho_1\,\rho'_3\,\rho_2\,((x:k) \to B) = \rho_1\,\rho_2\,\rho'_3\,((x:k) \to B)$, and by Lemma 5.6.7, $\cdot \vdash^{\mathsf{L}} \lambda x\!:\!k'.b : \rho_1\,\rho_2\,\rho_3\,((x:k) \to B)$ iff $\cdot \vdash^{\mathsf{L}} \lambda x\!:\!k'.b : \rho_1\,\rho_2\,\rho'_3\,((x:k) \to B)$. So, it will be enough to show that, for any $i \le j$:

1) $\mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho_1\,\rho_2\,\rho_3} = \mathcal{V}_{\mathsf{up}(\rho_2\,k)}[\![\rho_2\,k]\!]^{j}_{\rho_1\,\rho_3'}$

2) For any $V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho_1\,\rho_2\,\rho_3}$:

$$\mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^{i}_{\rho_1\,\rho_2\,\rho_3[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]} = \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(\rho_2\,B)}[\![\rho_2\,B]\!]^{i}_{\rho_1\,\rho_3'[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}$$

First, observe that 1) is exactly the IH for $(\mathsf{up}(k), i, \mathcal{F}_1, \mathcal{V})$ (using Lemma 5.6.6 to satisfy the requirement that $\rho_1\,\rho_2\,\rho_3 \cong^{\Gamma}_{i} \rho_1\,\rho_2\,\rho_3'$).

For 2), we begin by showing that, for any $V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^{i}_{\rho_1\,\rho_2\,\rho_3}$,

$$\rho_1\,\rho_2\,\rho_3[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})] \cong^{\Gamma,x:k}_{i} \rho_1\,\rho_2\,\rho_3'[x\mapsto(V,\mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]$$

To show this by EEqConsK, we must show that $\cdot \vdash V : \rho_1\,\rho_2\,\rho_3\,k$ and $\cdot \vdash V : \rho_1\,\rho_2\,\rho_3'\,k$ and that for any θ and $i' \leq i$, $\mathcal{V}^{\theta}_{\tau}[\![V]\!]^{i'}_{\emptyset} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k)}_{i'} \mathcal{V}^{\theta}_{\tau}[\![V]\!]^{i'}_{\emptyset}$. But we know by Lemma 5.5.17 that there is some derivation $\mathcal{E}'$ of $\cdot \vdash V : \rho_1\,\rho_2\,\rho_3\,k$. It follows by Lemma 5.6.7 that $\cdot \vdash V : \rho_1\,\rho_2\,\rho_3'\,k$. And that $\mathcal{V}^{\theta}_{\tau}[\![V]\!]^{i'}_{\emptyset}$ is related to itself is precisely the IH for $(\tau, i', \mathcal{E}', \mathcal{V})$.

Since these two environments are equivalent, 2) follows as the IH for $(\mathsf{up}(B), i, \mathcal{E}_2, \mathcal{C})$.

- The case for TArrComp is similar, but simpler.

- $$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma, x : k_1 \vdash B : k_2} \qquad \Gamma \vdash (x : k_1) \to k_2}{\Gamma \vdash \lambda x : k_1.B : (x : k_1) \to k_2}\ \text{TLamTLC}$$

  We must show that:

$$\mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1.B]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,((x:k_1)\to k_2))}_{j} \mathcal{V}^{\theta}_{\pi}[\![\lambda x : \rho_2\,k_1.\rho_2\,B]\!]^{j}_{\rho_1\,\rho_3'}$$

Expanding the definition of our equivalence and the interpretation, we find that it is sufficient to show that:

$$\begin{array}{l}
\forall A_1, A_2 \in \mathsf{CTyp}(\rho_1\,\rho_2\,\rho_3\,k_1), \forall \mathcal{I}_1', \mathcal{I}_2' \in [\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)]^{(\theta\times\mathbb{N})} \\
\quad A_1 \Rrightarrow^* V \wedge A_2 \Rrightarrow^* V \\
\to \forall i \leq j, \forall\theta, \mathcal{I}_1'(\theta, i) \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![A_1]\!]^{i}_{\emptyset} \\
\to \forall i \leq j, \forall\theta, \mathcal{I}_2'(\theta, i) \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![A_2]\!]^{i}_{\emptyset} \\
\to \forall i \leq j, \forall\theta, \mathcal{I}_1'(\theta, i) \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)}_{i} \mathcal{I}_2'(\theta, i) \\
\to \forall i \leq j, \qquad \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho_1\,\rho_2\,\rho_3[x\mapsto(A_1,\mathcal{I}_1')]} \\
\qquad\qquad \cong^{[V/x]\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_2)}_{i} \mathcal{V}^{\theta}_{\pi}[\![\rho_2\,B]\!]^{j}_{\rho_1\,\rho_3'[x\mapsto(A_2,\mathcal{I}_2')]}
\end{array}$$

So let some $A_1, A_2, \mathcal{I}_1'$ and $\mathcal{I}_2'$ be given that satisfy the assumptions above, and suppose $i \leq j$. By EEqConsK and Lemma 5.6.6, we have $\rho_1\,\rho_2\,\rho_3[x \mapsto (A_1, \mathcal{I}_1')] \cong^{\Gamma,x:k_1}_{i} \rho_1\,\rho_2\,\rho_3'[x \mapsto (A_2, \mathcal{I}_2')]$. Since $\pi = \mathsf{up}(\lambda x : k_1.B) = \mathsf{up}(B)$ by definition, the IH for $(\pi, i, \mathcal{E}_1, \mathcal{V})$ yields the desired result.

- The case for TLamDep is similar, but simpler.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \Gamma \vdash [A/x]k_2}{\Gamma \vdash B\ A : [A/x]k_2}$ TAppTLC

  Observing that $\mathsf{up}(B) \leq \mathsf{up}(B\ A)$ by definition, the IH for $(\mathsf{up}(B), j, \mathcal{E}_1, \mathcal{V})$ yields that:

$$\mathcal{V}^{\theta}_{\mathsf{up}(B)}[\![B]\!]^{j}_{\rho_1\,\rho_2\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,((x:k_1)\to k_2))}_{j} \mathcal{V}^{\theta}_{\mathsf{up}(B)}[\![\rho_2\,B]\!]^{j}_{\rho_1\,\rho_3'}$$

It follows by the definition of the equivalence that either both interpretations are the empty set or they are both set-theoretic functions of type:

$$(\mathsf{TYP} \times [\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)]^{(\theta\times\mathbb{N})}) \to [\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)]$$

In the former case, the desired result is immediate since $\emptyset \cong^{U}_{j} \emptyset$ for any U and j. So suppose they are both functions.

By confluence (Lemma 5.2.17) and because $\rho_1\,\rho_2\,\rho_3 \cong^{\Gamma}_{j} \rho_1\,\rho_2\,\rho_3'$, either $\rho_1\,\rho_2\,\rho_3\,A$ and $\rho_1\,\rho_2\,\rho_3'\,A$ parallel reduce to a common value V or neither parallel reduces to a value at all. In the later case, the interpretation of $B\ A$ is the empty set, so assume we have V. Then, by Lemma 5.5.5 and the definition of the interpretation, it is sufficient to show that:

$$\begin{aligned}&\mathcal{V}^{\theta}_{\mathsf{up}(B)}[\![B]\!]^{j}_{\rho_1\,\rho_2\,\rho_3}(\rho_1\,\rho_2\,\rho_3\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho_1\,\rho_2\,\rho_3})\\ \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,[A/x]k_2)}_{j}&\mathcal{V}^{\theta}_{\mathsf{up}(B)}[\![\rho_2\,B]\!]^{j}_{\rho_1\,\rho_3'}(\rho_1\,\rho_3'\,\rho_2\,A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![\rho_2\,A]\!]^{[0..j]}_{\rho_1\,\rho_3'})\end{aligned}$$

This is precisely the conclusion of the IH for $\mathcal{E}_1$ if we can show that:

1) $\forall i \leq j, \forall \theta, \mathcal{V}^{\theta}_{\tau}[\![A]\!]^{i}_{\rho_1\,\rho_2\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_1\,\rho_2\,\rho_3\,A]\!]^{i}_{\emptyset}$
2) $\forall i \leq j, \forall \theta, \mathcal{V}^{\theta}_{\tau}[\![\rho_2\,A]\!]^{i}_{\rho_1\,\rho_3'} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_1\,\rho_3'\,\rho_2\,A]\!]^{i}_{\emptyset}$
3) $\forall i \leq j, \forall \theta, \mathcal{V}^{\theta}_{\tau}[\![A]\!]^{i}_{\rho_1\,\rho_2\,\rho_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_2\,A]\!]^{i}_{\rho_1\,\rho_3'}$

Note that the way we split the environment into ρ_1, ρ_2, and ρ_3 is not fixed by the IH. We will show each of these using the IH for $\mathcal{E}_2$.

1) This is directly an instance of the IH for $(\tau, i, \mathcal{E}_2, \mathcal{V})$, using Lemma 5.6.6 and picking the "ρ_2" in our use of the IH to be $\rho_1\,\rho_2\,\rho_3$.
2) This is slightly trickier, but observe that we may use the IH for $\mathcal{E}_2$ with the environment split in two different ways to obtain:

$$\mathcal{V}^{\theta}_{\tau}[\![A]\!]^{i}_{\rho_1\,\rho_2\,\rho_3'} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho_3'\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_2\,A]\!]^{i}_{\rho_1\,\rho_3'}$$

and
$$\mathcal{V}^{\theta}_{\tau}[\![A]\!]^{i}_{\rho_1\,\rho_2\,\rho'_3} \cong^{\mathsf{Cand}(\rho_1\,\rho_2\,\rho'_3\,k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_1\,\rho_2\,\rho'_3\,A]\!]^{i}_{\emptyset}$$

The desired result then follows by the symmetry and transitivity of $\cong$ (Lemmas 5.6.2 and 5.6.3.

3) Similar to 1).

- Case TAppDep is similar to the previous case, but simpler.
- The remaining cases are relatively straightforward by induction. Cases TConv and TConvT require the use of Lemmas 5.6.4 and 5.6.5, respectively.

□

Lemma 5.7.2 ($\cong$ is reflexive)**.** Suppose $\rho \cong^{\Gamma}_{j} \rho'$.

- If $\Gamma \vdash A : k$ then $\Omega^{\theta}_{\pi}[\![A]\!]^{j}_{\rho} \cong^{\mathsf{Cand}(\rho\,k)}_{j} \Omega^{\theta}_{\pi}[\![A]\!]^{j}_{\rho'}$.
- If $\Gamma \vdash k$ then $\Omega_{\pi}[\![k]\!]^{j}_{\rho} \cong^{\mathsf{Cand}(\rho\,k)}_{j} \Omega_{\pi}[\![k]\!]^{j}_{\rho'}$.

Proof. A direct consequence of Lemma 5.7.1, picking $\rho_1 = \rho_2 = \emptyset$, and $\rho_3 = \rho$ and $\rho'_3 = \rho'$. □

Before we can prove the lemma relating the interpretation and reduction, we need one simple fact about environments and substitution. Because we have allowed types that are not values into our environments, it is not the case that the result of substituting an environment into a type value yields a type value. However, our well-formedness judgements on environments (like EnvWT and $\cong^{\Gamma}_{j}$) require all bound types to *reduce* to a value. Thus, we can prove the following weaker result:

Lemma 5.7.3. If $\mathsf{EnvWT}(\rho, \Gamma)$, then for any value V, there is another value V' such that $\rho\, V \Rrightarrow^{*} V'$.

Proof. By induction on the structure of values, observing in the variable case that $\mathsf{EnvWT}(\rho, \Gamma)$ ensures that any type variable bound by ρ parallel reduces to a value. □

We are now prepared to prove that taking a step of parallel reduction does not change the interpretation of a type or kind.

Lemma 5.7.4 (The interpretation respects reduction)**.** Let some π and j be given. Suppose we have a proof $\mathcal{E} :: \Gamma \vdash A : k$ or a proof $\mathcal{F} :: \Gamma \vdash k$, and let some Ω be given. If $\rho_1 \cong^{\Gamma}_{j} \rho_2$, then:

- If we have a proof $\mathcal{E} :: \Gamma \vdash A : k$ and $\pi = \mathsf{up}(A)$, then, if $A \Rrightarrow B$, then
  - If $\Omega = \mathcal{V}$, then $\mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^{j}_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\,k)}_{j} \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![B]\!]^{j}_{\rho_2}$.

– If $\Omega = \mathcal{C}$ and $k = s$, then $\mathcal{C}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^{j}_{\rho_1} = \mathcal{C}^{\theta}_{\mathsf{up}(A)}[\![B]\!]^{j}_{\rho_2}$.

- If we have a proof $\mathcal{F} :: \Gamma \vdash k$ and $\pi = \mathsf{up}(k)$, then, if $k \Rrightarrow k'$, then

$$\Omega_{\mathsf{up}(k)}[\![k]\!]^{j}_{\rho_1} = \Omega_{\mathsf{up}(k)}[\![k']\!]^{j}_{\rho_2}.$$

Proof. As in the previous proof, we proceed by lexicographic induction on the quadruple $(\pi, j, \mathcal{D}/\mathcal{E}, \Omega)$.

Note that, by Lemma 5.3.4 $\mathsf{up}(A) = \mathsf{up}(B)$ in the cases where $\mathcal{E} :: \Gamma \vdash A : k$, and $\mathsf{up}(k) = \mathsf{up}(k')$ in the cases where $\mathcal{F} :: \Gamma \vdash k$. We will use these facts repeatedly and implicitly in the remainder of the proof, to avoid getting bogged down with dozens of references to these two lemmas.

Now, Ω is either $\mathcal{C}$ or $\mathcal{V}$. The cases where $\Omega = \mathcal{C}$ are immediate by induction. For the cases where $\Omega = \mathcal{V}$, we consider the possible derivations of $\mathcal{E}$ or $\mathcal{F}$.

The cases for KSort, TVar and TNat are immediate because the subject can step only to itself, and by Lemma 5.7.2, the equivalence relation is reflexive (for the interpretations of types and kinds in the appropriate typing relations). In the remaining cases we ignore PTRefl and PKRefl when describing the possible ways a type or kind may take a step of parallel reduction, since the argument is always the same.

- $\mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash k_1} \qquad \overset{\mathcal{F}_2}{\Gamma, x : k_1 \vdash k_2}}{\Gamma \vdash (x : k_1) \to k_2}$ KArrTLC

  Inverting the definition of parallel reduction, we see that if $(x : k_1) \to k_2$ steps, it must be to some $(x : k_1') \to k_2'$ such that $k_1 \Rrightarrow k_1'$ and $k_2 \Rrightarrow k_2'$. Since $\pi = \mathsf{up}((x : k_1) \to k_2)$, we know it is σ. Recall the definition of the interpretation for this case:

$$\begin{aligned}\mathcal{V}_{\sigma}\,[\![(x : k_1) \to k_2]\!]^{j}_{\rho 1} \quad = \quad & \\ \{\lambda x : k_1'.B \mid \quad & \cdot \vdash \lambda x : k_1'.B : \rho_1\,((x : k_1) \to k_2) \\ & \text{and } \forall i \le j,\, \forall V \in \mathcal{V}_{\mathsf{up}(k_1)}[\![k_1]\!]^{i}_{\rho_1}, \\ & \quad [V/x]B \in \mathcal{C}_{\mathsf{up}(k_2)}[\![k_2]\!]^{i}_{\rho_1[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}\}\end{aligned}$$

  We can see that it will be enough to show that $\mathcal{V}_{\mathsf{up}(k_1)}[\![k_1]\!]^{i}_{\rho_1} = \mathcal{V}_{\mathsf{up}(k_1')}[\![k_1']\!]^{i}_{\rho_2}$, and that for any V in this set:

$$\mathcal{C}_{\mathsf{up}(k_2)}[\![k_2]\!]^{i}_{\rho_1[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]} = \mathcal{C}_{\mathsf{up}(k_2)}[\![k_2']\!]^{i}_{\rho_2[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]}$$

  The former is precisely the IH for $\mathcal{F}_1$. The latter will follow from the IH for $\mathcal{F}_2$ if we can show that:

$$\rho_1[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})] \cong^{\Gamma, x:k_1}_{j} \rho_2[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![V]\!]^{[0..i]}_{\emptyset})]$$

By the definition of the interpretation of kinds, we know that $\cdot \vdash V : \rho\, k_1$. The desired result follows by EEqConsK, observing that the equivalence relation is reflexive for well-kinded types (Lemma 5.7.2).

- Cases KArrDep, TArrComp, TArrPoly, TMonoPoly, TSigma, TSum, TMu and TAt are similarly straightforward.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma, x : k_1 \vdash B : k_2} \qquad \Gamma \vdash (x : k_1) \to k_2}{\Gamma \vdash \lambda x : k_1 . B : (x : k_1) \to k_2}$ TLamTLC

  By inversion on the reduction relation, $\lambda x : k_1.B$ must step to some $\lambda x : k_1.B'$ such that $B \Rrightarrow B'$. We must show that:

$$\mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1.B]\!]^{j}_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\,((x:k_1)\to k_2))}_{j} \mathcal{V}^{\theta}_{\pi}[\![\lambda x : k_1.B']\!]^{j}_{\rho_2}$$

  Expanding the definition of our equivalence and the interpretation, we find that it is sufficient to show that:

$$\begin{array}{l} \forall A_1, A_2 \in \mathsf{CTyp}(\rho_1\, k_1), \forall \mathcal{I}'_1, \mathcal{I}'_2 \in [\mathsf{Cand}(\rho_1\, k_1)]^{(\theta \times \mathbb{N})} \\ \quad A_1 \Rrightarrow^* V \wedge A_2 \Rrightarrow^* V \\ \to \forall i \leq j, \forall \theta, \mathcal{I}'_1(\theta, i) \cong^{\mathsf{Cand}(\rho_1\, k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![A_1]\!]^{i}_{\emptyset} \\ \to \forall i \leq j, \forall \theta, \mathcal{I}'_2(\theta, i) \cong^{\mathsf{Cand}(\rho_1\, k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![A_2]\!]^{i}_{\emptyset} \\ \to \forall i \leq j, \forall \theta, \mathcal{I}'_1(\theta, i) \cong^{\mathsf{Cand}(\rho_1\, k_1)}_{i} \mathcal{I}'_2(\theta, i) \\ \to \forall i \leq j, \qquad \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho_1[x \mapsto (A_1, \mathcal{I}'_1)]} \\ \qquad \cong^{[V/x]\mathsf{Cand}(\rho_1\, k_2)}_{i} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho_2[x \mapsto (A_2, \mathcal{I}'_2)]} \end{array}$$

  So let some $A_1, A_2, \mathcal{I}'_1$ and $\mathcal{I}'_2$ be given that satisfy the assumptions above, and suppose $i \leq j$. By EEqConsK and Lemma 5.6.6, we have $\rho_1[x \mapsto (A_1, \mathcal{I}'_1)] \cong^{\Gamma, x:k_1}_{i} \rho_2[x \mapsto (A_2, \mathcal{I}'_2)]$. Since $\pi = \mathsf{up}(\lambda x : k_1.B) = \mathsf{up}(B)$ by definition, the IH for $(\pi, i, \mathcal{E}_1, \mathcal{V})$ yields the desired result.

- Case TLamComp is similar.

- $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash B : (x : k_1) \to k_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash A : k_1} \qquad \Gamma \vdash [A/x]k_2}{\Gamma \vdash B\ A : [A/x]k_2}$ TAppTLC

  The type $B\ A$ may step in two ways. Consider each separately:

  - Suppose $B\ A \Rrightarrow B'\ A'$ by PTAppT1. We know that $B \Rrightarrow B'$ and $A \Rrightarrow A'$.

    Observing that $\mathsf{up}(B) \leq \mathsf{up}(B\ A)$ by definition, the IH for $(\mathsf{up}(B), j, \mathcal{E}_1, \mathcal{V})$ yields:

$$\mathcal{V}^{\theta}_{\mathsf{up}(B)}[\![B]\!]^{j}_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\,((x:k_1)\to k_2))}_{j} \mathcal{V}^{\theta}_{\mathsf{up}(B')}[\![B']\!]^{j}_{\rho_2}$$

    It follows by the definition of the equivalence that either both interpretations are the empty set, or they are both set-theoretic functions of type $(\mathsf{TYP} \times$

$[\mathsf{Cand}(\rho_1\, k_1)]^{(\theta\times\mathbb{N})}) \to [\mathsf{Cand}(\rho_1\, k_1)]$. In the former case, the desired result is immediate since $\emptyset \cong^U_j \emptyset$ for any U and j. So suppose they are both functions.

By confluence (Lemma 5.2.17) and because $\rho_1 \cong^\Gamma_j \rho_2$, either $\rho_1\, A$ and $\rho_2\, A'$ parallel reduce to a common value V or neither parallel reduces to a value at all. In the later case, the interpretation of $B\ A$ is the empty set, so assume we have V. Then, by Lemma 5.5.5 and the definition of the interpretation, it is sufficient to show that:

$$\begin{aligned}&\mathcal{V}^\theta_{\mathsf{up}(B)}[\![B]\!]^j_{\rho_1}(\rho_1\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_{\rho_1})\\ \cong^{\mathsf{Cand}(\rho_1\,[A/x]k_2)}_j &\mathcal{V}^\theta_{\mathsf{up}(B)}[\![B]\!]^j_{\rho_2}(\rho_2\, A', \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A']\!]^{[0..j]}_{\rho_2})\end{aligned}$$

This is precisely the conclusion of the IH for $\mathcal{E}_1$ if we can show that:

1) $\forall i \le j, \forall\theta, \mathcal{V}^\theta_\tau[\![A]\!]^i_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\, k_1)}_i \mathcal{V}^\theta_\tau[\![\rho_1\, A]\!]^i_\emptyset$

2) $\forall i \le j, \forall\theta, \mathcal{V}^\theta_\tau[\![A']\!]^i_{\rho_2} \cong^{\mathsf{Cand}(\rho_1\, k_1)}_i \mathcal{V}^\theta_\tau[\![\rho_2\, A']\!]^i_\emptyset$

3) $\forall i \le j, \forall\theta, \mathcal{V}^\theta_\tau[\![A]\!]^i_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\, k_1)}_i \mathcal{V}^\theta_\tau[\![A']\!]^i_{\rho_2}$

The first two of these are an instance of environment shifting (Lemma 5.7.1), while the third is precisely the IH for $\mathcal{E}_2$.

– Suppose instead that $B\ A$ steps by PTBetaT. We we know that B is $\lambda x : k_1'.B_1$ for some k_1' and B_1, that A is a value, and we have:

$$B_1 \Rrightarrow B_1'$$

$$A \Rrightarrow A'$$

$$B\ A \Rrightarrow [A'/x]B_1'$$

We must show that $\mathcal{V}^\theta_\pi[\![(\lambda x : k_1'.B_1)\ A]\!]^j_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\,[A/x]k_2)}_j \mathcal{V}^\theta_\pi[\![[A'/x]B_1']\!]^j_{\rho_2}$. By definition, we know that:

$$\mathcal{V}^\theta_\pi[\![(\lambda x : k_1'.B_1)\ A]\!]^j_{\rho_1} = \begin{cases}\mathcal{V}^\theta_\pi[\![(\lambda x : k_1'.B_1)]\!]^j_{\rho_1}\ (\rho_1\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_{\rho_1}) \\ \quad \text{If } \rho_1\, A \leadsto^* V \text{ and } (\rho_1\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j]}_{\rho_1}) \text{ is} \\ \quad\quad \text{in the domain of } \mathcal{V}^\theta_\pi[\![\lambda x : k_1'.B_1]\!]^j_{\rho_1} \\ \emptyset \quad \text{otherwise}\end{cases}$$

The first question is which of these cases applies. We know A is a value since the step occurred by PTBetaT, so by Lemmas 5.7.3 and 5.2.19 $\rho_1\, A \leadsto^* V$ for some V. To see that the pair mentioned in the case here is in the domain of the appropriate function, unfold the definition of the interpretation again to obtain:

$$\mathcal{V}^\theta_\pi[\![\lambda x : k_1'.B_1]\!]^j_{\rho_1} = (A \in \mathsf{TYP}, \mathcal{I} \in [\mathsf{Cand}(k_1')]^{(\theta\times\mathbb{N})}) \mapsto \mathcal{V}^\theta_\pi[\![B_1]\!]^j_{\rho_1[x\mapsto(A,\mathcal{I})]}$$

So to see that the first case above applies, it will be enough to show that, for any θ' and i, $\mathcal{V}^{\theta'}_{\tau}[\![A]\!]^{i}_{\rho_1} \in [\mathsf{Cand}(k_1')]$. But since the interpretation respects Cand (Lemma 5.5.5) we know that $\mathcal{V}^{\theta'}_{\tau}[\![A]\!]^{i}_{\rho_1} \in [\mathsf{Cand}(\rho_1\, k_1)]$ for any θ' and i. And by Lemma 5.4.6, $[\mathsf{Cand}(\rho_1\, k_1)] = [\mathsf{Cand}(\rho_1\, k_1')]$.

So, the first case of the definition of the interpretation for application applies. We know by Lemma 5.3.1 and regularity (Lemma 5.2.25) that $\pi = \mathsf{up}(B\ A) = \mathsf{up}(A) = \mathsf{up}(B) = \tau$. So, we must show that:

$$\mathcal{V}^{\theta}_{\tau}[\![B_1]\!]^{j}_{\rho_1[x \mapsto (\rho_1\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho_1})]} \cong^{\mathsf{Cand}(\rho_1\,[A/x]k_2)}_{j} \mathcal{V}^{\theta}_{\tau}[\![[A'/x]B_1']\!]^{j}_{\rho_2}.$$

Now, the IH for $(\tau, j, \mathcal{E}_1, \mathcal{V})$ will yield:

$$\mathcal{V}^{\theta}_{\tau}[\![B_1]\!]^{j}_{\rho_1[x \mapsto (\rho_1\, A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A]\!]^{[0..j]}_{\rho_1})]} \cong^{\mathsf{Cand}(\rho_1\,[A/x]k_2)}_{j} \mathcal{V}^{\theta}_{\tau}[\![B_1']\!]^{j}_{\rho_2[x \mapsto (\rho_2\, A', \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A']\!]^{[0..j]}_{\rho_2})]}$$

If we can show that:

1) $\forall i \leq j, \forall \theta, \mathcal{V}^{\theta}_{\tau}[\![A]\!]^{i}_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\, k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_1\, A]\!]^{i}_{\emptyset}$

2) $\forall i \leq j, \forall \theta, \mathcal{V}^{\theta}_{\tau}[\![A']\!]^{i}_{\rho_2} \cong^{\mathsf{Cand}(\rho_1\, k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![\rho_2\, A']\!]^{i}_{\emptyset}$

3) $\forall i \leq j, \forall \theta, \mathcal{V}^{\theta}_{\tau}[\![A]\!]^{i}_{\rho_1} \cong^{\mathsf{Cand}(\rho_1\, k_1)}_{i} \mathcal{V}^{\theta}_{\tau}[\![A']\!]^{i}_{\rho_2}$

The first two of these are an instance of environment shifting (Lemma 5.7.1), while the third is precisely the IH for $(\tau, i, \mathcal{E}_2, \mathcal{V})$.

To conclude, observe that:

$$\begin{aligned}
&\mathcal{V}^{\theta}_{\tau}[\![B_1']\!]^{j}_{\rho_2[x \mapsto (\rho_2\, A', \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A']\!]^{[0..j]}_{\rho_2})]} \\
&\quad \cong^{\mathsf{Cand}(\rho_2\,[A'/x]k_2)}_{j} \mathcal{V}^{\theta}_{\tau}[\![(\rho_2[x \mapsto (\rho_2\, A', \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_{\tau}[\![A']\!]^{[0..j]}_{\rho_2})])\, B_1']\!]^{j}_{\emptyset} && \text{(Lemma 5.7.1)} \\
&\quad \cong^{\mathsf{Cand}(\rho_2\,[A'/x]k_2)}_{j} \mathcal{V}^{\theta}_{\tau}[\![\rho_2\,[A'/x]B_1']\!]^{j}_{\emptyset} && \text{(Substitution)} \\
&\quad \cong^{\mathsf{Cand}(\rho_2\,[A'/x]k_2)}_{j} \mathcal{V}^{\theta}_{\tau}[\![[A'/x]B_1']\!]^{j}_{\rho_2} && \text{(Lemma 5.7.1)}
\end{aligned}$$

But by Lemma 5.6.4, Lemma 5.6.5, and the definition of $\cong^{\Gamma}_{j}$, we know that $\cong^{\mathsf{Cand}(\rho_1\,[A/x]k_2)}_{j}$ and $\cong^{\mathsf{Cand}(\rho_2\,[A'/x]k_2)}_{j}$ are the same relation. The desired result then follows by the transitivity of the equivalence relation (Lemma 5.6.3).

- Case TAppDep is similar (but simpler).

- $\mathcal{E} = \dfrac{\Gamma \vdash^{\mathsf{P}} a_1 : A \qquad \Gamma \vdash^{\mathsf{P}} a_2 : B}{\Gamma \vdash a_1 = a_2 : \star_{\tau}}$ TEq

By inversion on the definition of $\Rrightarrow$, we can see that if $a_1 = a_2$ steps it must be to $a_1' = a_2'$ for some a_1' and a_2' such that $a_1 \Rrightarrow a_1'$ and $a_2 \Rrightarrow a_2'$. According to the definition of the interpretation and our equivalence relation, we must show that:

$$\begin{array}{rl} & \{\mathsf{refl} \mid \cdot \vdash \rho_1\,(a_1 = a_2) : \star_\tau \text{ and } \exists b, \rho_1\, a_1 \Rrightarrow^* b \text{ and } \rho_1\, a_2 \Rrightarrow^* b\} \\ = & \{\mathsf{refl} \mid \cdot \vdash \rho_2\,(a_1' = a_2') : \star_\tau \text{ and } \exists b, \rho_2\, a_1' \Rrightarrow^* b \text{ and } \rho_2\, a_2' \Rrightarrow^* b\} \end{array}$$

Now, by preservation (Theorem 5.2.52) and environment substitution (Lemma 5.5.7), both $\rho_1\,(a_1 = a_2)$ and $\rho_2\,(a_1' = a_2')$ have kind $\star_\tau$. So, it will be enough to show that $\rho_1\, a_1$ and $\rho_1\, a_2$ reduce to a common term iff $\rho_2\, a_1'$ and $\rho_2\, a_2'$ do.

This will follow from confluence if we can show that $\rho_1\, a_1$ and $\rho_2\, a_1'$ reduce to a common term, and that $\rho_1\, a_2$ and $\rho_2\, a_2'$ reduce to a common term. We consider only the first pair—the other is similar.

By induction on the proof of $\rho_1 \cong_j^\Gamma \rho_2$, we find that $\rho_1\, a_1'$ and $\rho_2\, a_1'$ reduce to a common term. But by repeated application of Lemmas 5.2.15 and 5.2.16, we know that $\rho_1\, a_1 \Rrightarrow^* \rho_1\, a_1'$. Thus, $\rho_1\, a_1$ and $\rho_2\, a_1'$ reduce to a common term as required.

- Case TEqT is similar to the previous case.

□

5.8 The Fundamental Theorem, Normalization, and Consistency

We are almost prepared to prove the main soundness theorem for the interpretation. As we have done for several other lemmas in this chapter, we begin by defining a relation on environments. In particular, the main soundness result will only hold if the environment maps variables to terms or types in the interpretations of the appropriate types and kinds. Intuitively, if $\rho \models_j \Gamma$, then ρ is a model of Γ.

$$\frac{}{\emptyset \models_j \cdot}\ \textsc{EnvNil} \qquad \frac{\begin{array}{l} \cdot \vdash^{\mathsf{L}} v : \rho\, A \\ v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^j_\rho \\ \rho \models_j \Gamma \end{array}}{\rho[x \mapsto v] \models_j \Gamma, x : A}\ \textsc{EnvConsT}$$

$$\frac{\begin{array}{l} \cdot \vdash A : \rho\, k \\ A \in \mathcal{C}_{\mathsf{up}(k)}[\![k]\!]^j_\rho \\ \rho \models_j \Gamma \quad j \leq j' \end{array}}{\rho[x \mapsto (A, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![A]\!]^{[0..j']}_\emptyset)] \models_j \Gamma, x : k}\ \textsc{EnvConsK}$$

We need several lemmas about this relation. To begin, recall that we have previously defined a number of other relations between environments and context (like $\mathsf{EnvTyp}(\rho,\Gamma)$). We have defined $\rho \models_j \Gamma$ to be a subrelation of each of the previous relations. This means we can use it as evidence for, for example, $\mathsf{EnvTyp}(\rho,\Gamma)$ or $\mathsf{EnvCand}(\rho,\Gamma)$. To avoid getting bogged down in notation in the proofs below, we will usually elide the step of reasoning that converts from a proof of $\rho \models_j \Gamma$ to one of these other relations when referring a lemma that depends on them.

Lemma 5.8.1. If $\rho \models_j \Gamma$ then $\mathsf{EnvTyp}(\rho,\Gamma)$.

Proof. By induction on the proof of $\rho \models_j \Gamma$. □

Lemma 5.8.2. If $\rho \models_j \Gamma$ then $\rho \cong_j^\Gamma \rho$.

Proof. By induction on the proof of $\rho \models_j \Gamma$, using Lemma 5.7.2. □

Lemma 5.8.3. If $\rho \models_j \Gamma$ then $\rho \sqsubseteq_{\mathsf{DC}}^\Gamma \rho$.

Proof. By induction on the proof of $\rho \models_j \Gamma$, using Lemma 5.5.21 in the EnvConsK case. □

Lemma 5.8.4. If $\rho \models_j \Gamma$ then $\rho \sqsubseteq_{\mathsf{LP}}^\Gamma \rho$.

Proof. By induction on the proof of $\rho \models_j \Gamma$, using Lemma 5.5.24 in the EnvConsK case. □

We will also need a version of the downward closure lemma for this new environment soundness judgement:

Lemma 5.8.5 (Downward Closure for Environments)**.** If $j_1 \leq j_2$ and $\rho \models_{j_2} \Gamma$, then $\rho \models_{j_1} \Gamma$.

Proof. By induction on the proof of $\rho \models_{j_2} \Gamma$, using Lemmas 5.5.22 and 5.5.23. □

The next four lemmas essentially capture the variable case of the main soundness theorem.

Lemma 5.8.6 (Environment inversion (terms, $\mathcal{V}$))**.** Suppose $\rho \models_j \Gamma$ and $\vdash \Gamma$. If $(x : A) \in \Gamma$, then $\rho\, x \in \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$.

Proof. By induction on the derivation of $\rho \models_j \Gamma$. The case EnvNil does not apply. We consider the remaining two cases individually:

- Suppose the derivation goes by EnvConsT:

$$\frac{\begin{array}{c}\cdot \vdash^{\mathsf{L}} v : \rho'\, B \\ v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^j_{\rho'} \\ \rho' \models_j \Gamma\end{array}}{\rho'[y \mapsto v] \models_j \Gamma, y : B}\ \textsc{EnvConsT}$$

Either $x = y$ or not.

– Suppose $x = y$. Then B in the rule instance above is A from the theorem statement, and $\rho\, x$ is v. So the second premise of the rule reads:

$$\rho\, x \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho'}$$

This differs from our desired conclusion in two ways. First, θ may not be L. But since $\vdash \Gamma$, we know that $\mathsf{Mob}\,(A)$, and thus by Lemma 5.5.9, we have

$$\rho\, x \in \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho'}$$

Second, ρ' is not quite ρ. But since $\vdash \Gamma$, we know x does not appear free in A. Thus, by the weakening/strengthening lemma for the interpretation (Lemma 5.5.10), we have $\rho\, x \in \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho}$ as desired.

– Suppose $x \neq y$. Observe that $\rho\, x = \rho'\, x$ since $x \neq y$. Thus, the IH gives us that $\rho\, x \in \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho'}$.

The weakening/strengthening lemma for the interpretation (Lemma 5.5.10) gives us that $\mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho} = \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho'}$, concluding the case.

- The case for EnvConsK is similar to the previous subcase, since the last binding on the environment is not a term variable.

□

Lemma 5.8.7 (Environment type inversion (terms, $\mathcal{C}$))**.** Suppose $\rho \models_j \Gamma$ and $\vdash \Gamma$. If $(x : A) \in \Gamma$, then $\rho\, x \in \mathcal{C}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho}$.

Proof. By Lemma 5.8.6, we have $\rho\, x \in \mathcal{V}^{\theta}_{\mathsf{up}(A)}[\![A]\!]^j_{\rho}$. By the definition of the interpretation, it will therefore be enough to show that $\cdot \vdash^{\theta} \rho\, x : \rho\, A$. But this follows by Lemma 5.5.7, observing that $\Gamma \vdash^{\theta} x : A$ by TVar. □

Lemma 5.8.8 (Environment inversion (types, $\mathcal{C}$))**.** If $\rho \models_j \Gamma$ and $(x : k) \in \Gamma$ then $\rho\, x \in \mathcal{C}_{\mathsf{up}(k)}[\![k]\!]^j_{\rho}$.

Proof. Similar to the proof of Lemma 5.8.6, but simpler since there is no logicality to consider. □

We are now ready to prove the fundamental theorem of the interpretation. This demonstrates that the interpretation is a good model of what can be typed in the system. As we will see later, it implies that everything in the logical fragment normalizes.

Theorem 5.8.9 (Fundamental Theorem of the Interpretation)**.** Suppose $\rho \models_j \Gamma$.

- If $\mathcal{D} :: \Gamma \vdash^{\theta_1} a : A$, then $\rho\, a \in \mathcal{C}^{\theta_1}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$.
- If $\mathcal{E} :: \Gamma \vdash A : k$, then $\rho\, A \in \mathcal{C}_{\mathsf{up}(k)}[\![k]\!]^j_\rho$.

Proof. By mutual induction on the derivations $\mathcal{D}$ and $\mathcal{E}$. Note that, in particular, we leave ρ and j general so that we may instantiate them however we choose when applying an IH. Consider the possible forms of each derivation.

- $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma \vdash^{\mathsf{L}} a_1 : B_1 = B_2 \\ \mathsf{hd}\,(B_1) = hf_1 \qquad \mathsf{hd}\,(B_2) = hf_2 \qquad hf_1 \neq hf_2 \\ \Gamma \vdash^{\theta_2} a : A \qquad \Gamma \vdash A : \star_\sigma \qquad \Gamma \vdash B : \star_\sigma\end{array}}{\Gamma \vdash^{\theta_1} a : B}\ \textsc{EContra}$$

  In this case we will find a contradiction.

  The IH for $\mathcal{D}_1$ and the definition of the interpretation yield that there exists some term B_3 such that $\rho\, B_1 \Rrightarrow^* B_3$ and $\rho\, B_2 \Rrightarrow^* B_3$.

  One of the derivation's hypotheses yields B_1 and B_2 both have head forms and that they are not the same. By Lemmas 5.5.2 and 5.2.10 we find that B_3 has the same head form as B_1. But by the same reasoning, B_3 must have the same head form as B_2. This is a contradiction, since it implies B_1 and B_2 have the same head forms.

- $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma \vdash^{\theta_1} a : (A_1 @ \theta_2) = (A_2 @ \theta_2) \qquad \Gamma \vdash A_1 = A_2 : \star_\tau\end{array}}{\Gamma \vdash^{\theta_1} a : A_1 = A_2}\ \textsc{EAtInv}$$

  We consider the cases for θ_1 separately.

  - Suppose $\theta_1 = \mathsf{L}$. Unfolding the definition of the interpretation, we must show $\rho\, a \leadsto^* \mathsf{refl}$ and that $\rho\, A_1$ and $\rho\, A_2$ parallel reduce to a common type.

    The IH for derivation $\mathcal{D}_1$ and the definition of the interpretation give us that $\rho\, a \leadsto^* \mathsf{refl}$ (as desired) and that there exists some type B such that $\rho\,(A_1 @ \theta_2) \Rrightarrow^* B$ and $\rho\,(A_2 @ \theta_2) \Rrightarrow^* B$. By Lemma 5.2.10, B must have the form $B' @ \theta_2$ for some B'. Thus, by Lemma 5.2.1, $A_1 \Rrightarrow^* B'$ and $A_2 \Rrightarrow B'$, concluding this case.

- Suppose $\theta_1 = \mathsf{P}$. Unfolding the definition of the interpretation, we must show that if $\rho\, a \rightsquigarrow^i v$ for some $i \leq j$, then $v = \mathsf{refl}$, and $\rho\, A_1$ and $\rho\, A_2$ parallel reduce to a common type.

  So, suppose $\rho\, a \rightsquigarrow^i v$ for some $i \leq j$. Unfolding the IH for $\mathcal{D}_1$ therefore gives us that $v = \mathsf{refl}$ (as desired) and that $\rho\,(A_1@\theta_2)$ and $\rho\,(A_2@\theta_2)$ parallel reduce to a common type. The result then follows by the same reasoning as in the previous subcase.

- Cases ESumInv1, ESumInv2, EArrInv1 and ESigmaInv1 are similar to the previous case.

- $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \vdash^{\theta_1} a : ((x : A_1) \to A_2) = ((x : B_1) \to B_2) \\ \Gamma \vdash^{\theta_2} v : A_1 \qquad \Gamma \vdash [v/x]A_2 = [v/x]B_2 : \star_\tau\end{array}}{\Gamma \vdash^{\theta_1} a : [v/x]A_2 = [v/x]B_2}\ \text{EArrInv2}$$

  We consider the cases for θ_1 separately.

  - Suppose $\theta_1 = \mathsf{L}$. Unfolding the definition of the interpretation, we must show $\rho\, a \rightsquigarrow^* \mathsf{refl}$ and that $\rho\,[v/x]A_2$ and $\rho\,[v/x]B_2$ parallel reduce to a common type.

    The IH for the derivation $\mathcal{D}_1$ and the definition of the interpretation give us that $\rho\, a \rightsquigarrow^* \mathsf{refl}$ (as desired) and that there exists some type A such that $\rho\,((x : A_1) \to A_2) \Rrightarrow^* A$ and $\rho\,((x : B_1) \to B_2) \Rrightarrow^* A$. By Lemma 5.2.10, A must have the form $(x : A_1') \to A_2'$. Thus, by Lemma 5.2.2, $\rho\, A_2 \Rrightarrow^* A_2'$ and $\rho\, B_2 \Rrightarrow^* A_2'$.

    By Lemma 5.2.15, it follows that $[\rho\, v/x]\rho\, A_2 \Rrightarrow^* [\rho\, v/x]A_2'$ and $[\rho\, v/x]\rho\, B_2 \Rrightarrow^* [\rho\, v/x]A_2'$. But $[\rho\, v/x]\rho\, A_2 = \rho\,[v/x]A_2$ and $[\rho\, v/x]\rho\, B_2 = \rho\,[v/x]B_2$. So $\rho\,[v/x]A_2$ and $\rho\,[v/x]B_2$ parallel reduce to a common expression, as desired.

  - Suppose $\theta_1 = \mathsf{P}$. Unfolding the definition of the interpretation, we must show that if $\rho\, a \rightsquigarrow^i v$ for some $i \leq j$, then $v = \mathsf{refl}$, and $\rho\,[v/x]A_2$ and $\rho\,[v/x]B_2$ parallel reduce to a common type.

    So suppose $\rho\, a \rightsquigarrow^i v$ for some $i \leq j$. Unfolding the IH for $\mathcal{D}_1$ therefore gives us that $v = \mathsf{refl}$ (as desired) and that $\rho\,((x : A_1) \to A_2)$ and $\rho\,((x : B_1) \to A_2)$ parallel reduce to a common type. The result then follows by the same reasoning as in the previous subcase.

- Cases ESigmaInv2, and EMuInv are similar to the previous case.

- $$\mathcal{D} = \cfrac{(x : A) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash^{\theta} x : A}\ \text{EVar}$$

  This case is immediate by Lemma 5.8.7.

- $\mathcal{D} = \dfrac{(x : k) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : k}$ TVar

  This case is immediate by lemma 5.8.8.

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\theta_2} v : A@\theta_1} \qquad \Gamma \vdash A : \star_\sigma}{\Gamma \vdash^{\theta_1} v : A}$ EUnboxVal

  By Lemma 5.5.6, $\rho\, v$ is a value. Thus, by the definition of the interpretation for either case of θ_1, it is enough to show that $\rho\, v \in \mathcal{V}^{\theta_1}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$.

  The IH for $\mathcal{D}_1$ and the definition of the interpretation yield that $\rho\, v \in \mathcal{V}^{\theta_2}_{\mathsf{up}(A@\theta_1)}[\![A@\theta_1]\!]^j_\rho$. The definitions of the interpretation and up for @-types yield $\mathcal{V}^{\theta_2}_{\mathsf{up}(A@\theta_1)}[\![A@\theta_1]\!]^j_\rho = \mathcal{V}^{\theta_1}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$, so $\rho\, v \in \mathcal{V}^{\theta_1}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$ as desired.

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\theta_1} b : (x : A) \to B} \qquad \overset{\mathcal{D}_2}{\Gamma \vdash^{\theta_1} a : A} \qquad \Gamma \vdash [a/x]B : \star_\sigma}{\Gamma \vdash^{\theta_1} b\ a : [a/x]B}$ EAppComp

  We consider the cases for θ_1 separately.

  – Suppose θ_1 is L. We must show that $\rho\,(b\ a) \leadsto^* v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}([a/x]B)}[\![[a/x]B]\!]^j_\rho$.

    The IH for $\mathcal{D}_2$ gives us that $\rho\, a \leadsto^* v' \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$. The IH for $\mathcal{D}_1$ gives us that $\rho\, b \leadsto^* \lambda x\!:\!A'.b'$ such that $\cdot \vdash^{\mathsf{L}} \lambda x\!:\!A'.b : (x : A) \to B$ and

$$\begin{aligned}&\forall i \leq j,\ \forall v' \in \mathcal{V}^{\mathsf{L}}_{\mathsf{min}(\mathsf{up}((x:A)\to B),\mathsf{up}(A))}[\![A]\!]^i_\rho,\\ &\qquad [v'/x]b' \in \mathcal{C}^{\mathsf{L}}_{\mathsf{min}(\mathsf{up}((x:A)\to B),\mathsf{up}(B))}[\![B]\!]^i_{\rho[x\mapsto v']}\end{aligned}$$

    Now, Lemmas 5.3.5 and 5.3.6 show that $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(A)) = \mathsf{up}(A)$ and $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(B)) = \mathsf{up}(B)$. Thus, combining the two IHs yields:

$$[v'/x]b' \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^j_{\rho[x\mapsto v']}$$

    By the definition of the interpretation, we therefore have:

$$\rho\,(b\ a) \leadsto^* [v'/x]b' \leadsto^* v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^j_{\rho[x\mapsto v']}$$

    So, it will be enough to show that $\mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^j_{\rho[x\mapsto v']} = \mathcal{V}^{\mathsf{L}}_{\mathsf{up}([a/x]B)}[\![[a/x]B]\!]^j_\rho$. Combining several lemmas achieves this result:

$$\begin{aligned}\mathcal{V}^{\mathsf{L}}_{\mathsf{up}([a/x]B)}[\![[a/x]B]\!]^j_\rho &= \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![[a/x]B]\!]^j_\rho && \text{(Lemma 5.3.2)}\\ &= \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![[\rho\, a/x]\rho\, B]\!]^j_\emptyset && \text{(Lemma 5.7.1)}\\ &= \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![[v'/x]\rho\, B]\!]^j_\emptyset && \text{(Lemma 5.7.4)}\\ &= \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^j_{\rho[x\mapsto v']} && \text{(Lemma 5.7.1)}\end{aligned}$$

Note that Lemmas 5.7.4 and 5.7.1 actually state a result in terms of $\cong_j^{\mathsf{Cand}(k)}$ where k is the kind of the type being interpreted. However, here we know $\Gamma \vdash [a/x]B : \star_\sigma$ by a hypotheses of EAppComp, so k is $\star_\sigma$, in which case $\cong$ is defined to be set equality.

These lemmas have several prerequisites which are largely straightforward to dispatch. We note in particular that the final use of Lemma 5.7.1 requires us to observe that $\Gamma, x : A \vdash B : s$ and that $\rho[x \mapsto v'] \cong_j^{\Gamma, x:A} \rho[x \mapsto v']$. The former follows by a use of regularity (Lemma 5.2.25) with $\mathcal{D}_1$, and inversion for arrow types (Lemma 5.2.41). For the latter, by Lemma 5.8.2 it is enough to show that $\rho[x \mapsto v'] \models_j \Gamma, x : A$. This is true by EnvConsT, the first hypothesis of which holds by substitution (Lemma 5.5.7) and preservation (Lemma 5.2.53), and the rest of which we have already satisfied.

– Suppose instead that θ_1 is P. We must show that, if $\rho\,(b\ \ a) \rightsquigarrow^i v$ for some $i \leq j$, then $v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}([a/x]B)}[\![[a/x]B]\!]^{(j-i)}_\rho$.

So, suppose $\rho\,(b\ \ a) \rightsquigarrow^i v$. The inversion lemma for reduction of application (5.2.3) has two cases, depending on whether $\rho\, b$ evaluates to a lambda or a recursive function. We show only the case for recursive functions—the other is similar but simpler.

In this case, the lemma yields $\rho\, b \rightsquigarrow^{i_1} \mathsf{rec}\ f\ (x : A').b'$ and $\rho\, a \rightsquigarrow^{i_2} v'$ for some b' and a' such that $[v'/x][\mathsf{rec}\ f\ (x : A').b'/f]b' \rightsquigarrow^{i_3} v$. Additionally, $1 + i_1 + i_2 + i_3 = i$.

The IH for $\mathcal{D}_2$ therefore yields $v' \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(A)}[\![A]\!]^{(j-i_2)}_\rho$. The IH for $\mathcal{D}_1$ gives us that:

$$\begin{aligned}&\forall i' < (j - i_1),\, \forall v' \in \mathcal{V}^{\mathsf{P}}_{\mathsf{min}(\mathsf{up}((x:A)\to B),\mathsf{up}(A))}[\![A]\!]^{i'}_\rho,\\ &\qquad\qquad [v'/x][\mathsf{rec}\, f\ (x\,{:}\,A').b'/f]b' \in \mathcal{C}^{\mathsf{P}}_{\mathsf{min}(\mathsf{up}((x:A)\to B),\mathsf{up}(B))}[\![B]\!]^{i'}_{\rho[x\mapsto v']}\end{aligned}$$

By downward closure (Lemma 5.5.22), we have:

$$v' \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(A)}[\![A]\!]^{(j-i_2-i_1-1)}_\rho$$

Additionally, Lemmas 5.3.5 and 5.3.6 show that $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(A))$ $= \mathsf{up}(A)$ and $\mathsf{min}(\mathsf{up}((x : A) \to B), \mathsf{up}(B)) = \mathsf{up}(B)$. Thus, combining the two IHs yields:

$$[v'/x][\mathsf{rec}\, f\ (x\,{:}\,A').b'/f]b' \in \mathcal{C}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{(j-i_2-i_1-1)}_{\rho[x\mapsto v']}$$

By the definition of the interpretation and recalling that $[v'/x][\mathsf{rec}\ f\ (x : A').b'/f]b' \rightsquigarrow^{i_3} v$, we therefore have:

$$v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{(j-(1+i_1+i_2+i_3))}_{\rho[x\mapsto v']}$$

So, recalling that $i = 1 + i_1 + i_2 + i_3$, it will be enough to show that:

$$\mathcal{V}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{(j-i)}_{\rho[x\mapsto v']} = \mathcal{V}^{\mathsf{P}}_{\mathsf{up}([a/x]B)}[\![[a/x]B]\!]^{(j-i)}_\rho$$

But, after observing that $\rho \models_{j-i} \Gamma$ by downward closure for environments Lemma 5.8.5, this follows by the same reasoning as in the previous subcase.

- Cases EAppPoly, TAppTLC and TAppDep are similar.

- $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x : k \vdash^{\mathsf{L}} b : B} \qquad \Gamma \vdash (x : k) \to B : \star_\sigma}{\Gamma \vdash^{\mathsf{L}} \lambda x{:}k.b : (x : k) \to B} \text{ ELamPoly}$$

  In this case, we must show that $\rho\,(\lambda x : k.b) \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}((x:k)\to B)}[\![(x : k) \to B]\!]^j_\rho$. Unfolding the definition of the interpretation, we find that we must show:

  $$\forall i \leq j,\ \forall V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^i_\rho,\ [V/x]\rho\, b \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(B)}[\![B]\!]^i_{\rho[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau [\![V]\!]^{[0..i]}_\emptyset)]}$$

  So let some $i \leq j$ and $V \in \mathcal{V}_{\mathsf{up}(k)}[\![k]\!]^i_\rho$ be given. By the IH for $\mathcal{D}_1$, it will be enough to show that $\rho[x \mapsto (V, \mathcal{V}^{\{\mathsf{L},\mathsf{P}\}}_\tau[\![V]\!]^{[0..i]}_\emptyset)] \models_i \Gamma, x : A$. By EnvConsK, it will be enough to show that $\cdot \vdash V : \rho\, k$ and $V \in \mathcal{C}_{\mathsf{up}(k)}[\![k]\!]^i_\rho$. The former holds by Lemma 5.5.18, and the latter follows by the definition of the computational interpretation, since we already know V is in the value interpretation of the same kind.

- Cases ELamComp, ERecComp, ERecPoly, TLamTLC and TLamDep are similar

- $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\mathsf{L}} a : A}}{\Gamma \vdash^{\mathsf{P}} a : A} \text{ ESub}$$

  The IH for $\mathcal{D}_1$ yields $\rho\, a \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$. By regularity (Lemma 5.2.25), we know that $\Gamma \vdash A : \star_\sigma$. The result follows by Lemma 5.5.25.

- The case for EMVal is straightforward by induction and Lemma 5.5.9.

- Cases EZero, ESucc, EPair, EInl, EInr, ERoll, ERefl and EReflT are straightforward by induction and the definition of the interpretation.

- $$\dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\theta} a : (\Sigma x{:}A_1 \,.\, A_2)@\theta'} \qquad \begin{array}{c}\overset{\mathcal{D}_2}{\Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a \vdash^{\theta} b : B} \\ \Gamma \vdash B : \star_\sigma\end{array}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b : B} \text{ EPCase}$$

  We show only the case for $\theta = \mathsf{P}$. The other case is similar, but simpler. We must show that if $\mathsf{pcase}_z\ \rho\, a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow \rho\, b \rightsquigarrow^{j'} v$ for some $j' \leq j$, then $v \in \mathcal{V}^{\mathsf{P}}_\pi[\![B]\!]^{j-j'}_\rho$.

So suppose $\mathsf{pcase}_z\ \rho\, a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow \rho\, b \leadsto^{j'} v$. They by Lemma 5.2.4, there are some i_1 and i_2 such that $1 + i_1 + i_2 = j'$ and:

$$\rho\, a \leadsto^{i_1} \langle v_1, v_2\rangle \qquad \text{and} \qquad [v_1/x][v_2/y][\mathsf{refl}/z]b \leadsto^{i_2} v$$

We must show that:

$$v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{j-j'}_{\rho}$$

Now, by the induction hypothesis for $\mathcal{D}_1$ and the definition of the interpretation, we have:

$$v_1 \in \mathcal{V}^{\theta'}_{\mathsf{up}(A_1)}[\![A_1]\!]^{j-j_1}_{\rho} \qquad \text{and} \qquad v_2 \in \mathcal{V}^{\theta'}_{\mathsf{up}(A_2)}[\![A_2]\!]^{j-j_1}_{\rho[x\mapsto v_1]}$$

Now, by regularity (Lemma 5.2.25), inversion for @-types (Lemma 5.2.40) and inversion for Σ-types (Lemma 5.2.42, we have $\mathsf{Mob}\,(A_1)$. It follows by Lemma 5.5.9 and the definition of the interpretation for @-types that:

$$v_1 \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A_1)}[\![A_1]\!]^{j-j_1}_{\rho} \qquad \text{and} \qquad v_2 \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A_2)}[\![A_2@\theta']\!]^{j-j_1}_{\rho[x\mapsto v_1]}$$

So, by two applications of EnvConsT and downward closure for environments (Lemma 5.8.5), we have:

$$\rho[x \mapsto v_1][y \mapsto v_2] \models_{j-j_1} \Gamma, x : A_1, y : A_2@\theta'$$

Additionally, by the definition of the interpretation, we have that:

$$\mathsf{refl} \in \mathcal{V}^{\mathsf{L}}_{\tau}[\![\langle x, y\rangle = a]\!]^{j-j_1}_{\rho[x\mapsto v_1][y\mapsto v_2]}$$

So it follows again by EnvConsT that and downward closure for environments that:

$$\rho[x \mapsto v_1][y \mapsto v_2][z \mapsto \mathsf{refl}] \models_{j-j_1-j_2} \Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a$$

Thus, the IH for $\mathcal{D}_2$ and downward closure (Lemma 5.5.22) yield:

$$v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(B)}[\![B]\!]^{j-(1+j_1+j_2)}_{\rho[x\mapsto v_1][y\mapsto v_2][z\mapsto \mathsf{refl}]}$$

But $1 + j1 + j2 = j'$, and we know that x, y and z are not free in B by the typing hypotheses to $\mathcal{D}$. So by Lemma 5.5.10 we obtain the desired result.

- Cases ENCase, ESCase, and EUnroll are similar.

- $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2} \qquad \overset{\mathcal{D}_2}{\Gamma \vdash^{\theta_1} a : [b_1/x]A} \qquad \Gamma \vdash [b_2/x]A : \star_\sigma}{\Gamma \vdash^{\theta_1} a : [b_2/x]A}\ \text{EConv}$$

We must show that $\rho\, a \in \mathcal{C}^{\theta_1}_{\mathsf{up}([b_2/x]A)}[\![[b_2/x]A]\!]^j_\rho$. The IH for $\mathcal{D}_2$ yields $\rho\, a \in \mathcal{C}^{\theta_1}_{\mathsf{up}([b_1/x]A)}[\![[b_2/x]A]\!]^j_\rho$. So it will be enough to show that:

$$\mathcal{C}^{\theta_1}_{\mathsf{up}([b_2/x]A)}[\![[b_2/x]A]\!]^j_\rho = \mathcal{C}^{\theta_1}_{\mathsf{up}([b_1/x]A)}[\![[b_2/x]A]\!]^j_\rho$$

Now, by the IH for $\mathcal{D}_1$ and the definition of the interpretation, there is some expression b such that $\rho\, b_1 \Rrightarrow^* b$ and $\rho\, b_2 \Rrightarrow^* b$. Also, observe that, by Lemmas 5.3.2, 5.3.3 and 5.5.8, $\mathsf{up}(A) = \mathsf{up}([b_1/x]A) = \mathsf{up}([b_2/x]A) = \mathsf{up}(\rho\,[b_1/x]A) = \mathsf{up}(\rho\,[b_2/x]A)$. We will refer to this polymorphism flag as π below.

So, we may prove the desired equality as follows:

$$\begin{aligned}
\mathcal{C}^{\theta_1}_{\pi}[\![[b_2/x]A]\!]^j_\rho &= \mathcal{C}^{\theta_1}_{\pi}[\![\rho\,[b_2/x]A]\!]^j_\emptyset && \text{(Lemma 5.7.1)}\\
&= \mathcal{C}^{\theta_1}_{\pi}[\![\rho\,[b/x]A]\!]^j_\emptyset && \text{(Lemma 5.7.4)}\\
&= \mathcal{C}^{\theta_1}_{\pi}[\![\rho\,[b_1/x]A]\!]^j_\emptyset && \text{(Lemma 5.7.4)}\\
&= \mathcal{C}^{\theta_1}_{\pi}[\![[b_1/x]A]\!]^j_\rho && \text{(Lemma 5.7.1)}
\end{aligned}$$

To justify the uses of Lemma 5.7.1, we must show that $\Gamma \vdash [b_1/x]A : s$ and $\Gamma \vdash [b_2/x]A : s$, which both follow from regularity (Lemma 5.2.25), with $\mathcal{D}$ and $\mathcal{D}_2$. The uses of Lemma 5.7.4 have several prerequisites. First, we must show that $\rho\,[b_1/x]A$ and $\rho\,[b_2/x]A$ are well-kinded, which follows from regularity (Lemma 5.2.25) and environment substitution (Lemma 5.5.7). Additionally, we must show that $\rho\,[b_2/x]A \Rrightarrow^* \rho\,[b/x]A$ and $\rho\,[b_1/x]A \Rrightarrow^* \rho\,[b/x]A$. This follows by Lemmas 5.2.13, 5.2.15 and 5.2.16.

- Cases EConvT, TConv, and TConvT are similar.

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\mathsf{P}} v : A}}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}$ EBoxP

  We must show that $\rho\, v \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(A@\mathsf{P})}[\![A@\mathsf{P}]\!]^j_\rho$. According to the definition of the interpretation and up, it will be enough to show that $\rho\, v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$. This follows immediately by the IH for $\mathcal{D}_2$ and the definition of the interpretation.

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash^{\mathsf{L}} a : A}}{\Gamma \vdash^{\mathsf{L}} a : A@\theta}$ EBoxL

  We must show that $\rho\, a \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(A@\mathsf{L})}[\![A@\theta]\!]^j_\rho$. If θ is L, this follows immediately by the IH for $\mathcal{D}_2$ and the definition of the interpretation and up.

  So suppose instead that θ is P. Since $\mathsf{up}(A@\theta) = \mathsf{up}(A)$, we will use the latter exclusively. The IH for $\mathcal{D}_2$ yields there is some v such that $\rho\, a \leadsto^* v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^j_\rho$.

We must show that if $\rho\, a \leadsto^i v$ for some $i \leq j$, then $v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(A)}[\![A]\!]^{(j-i)}_{\rho}$. But because the value interpretation models subsumption (Lemma 5.5.24), we know $v \in \mathcal{V}^{\mathsf{P}}_{\mathsf{up}(A)}[\![A]\!]^{j}_{\rho}$, and the result then follows by downward closure (Lemma 5.5.22).

- $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \vdash^{\theta_1} a : A\end{array}}{\Gamma \vdash^{\mathsf{P}} a : A@\theta_1}\ \textsc{EBoxP}$$

  If $\theta = \mathsf{P}$, the result is immediate by the definition of the interpretation and the induction hypothesis for $\mathcal{D}_1$. So suppose $\theta = \mathsf{L}$.

  By the definition of the interpretation and up, we must show that if $\rho\, a \leadsto^i v$ for some $i \leq j$, then $v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^{(j-i)}_{\rho}$. But by the IH for $\mathcal{D}_1$, we know that $v \in \mathcal{V}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^{j}_{\rho}$, so the result follows by downward closure (Lemma 5.5.22).

- $$\mathcal{E} = \dfrac{\begin{array}{l}\Gamma \vdash k \\ \Gamma, x : k \vdash B : \star_\sigma\end{array}}{\Gamma \vdash (x : k) \to B : \star_\sigma}\ \textsc{TArrPoly}.$$

  In this case, we must show that $\rho\,((x : k) \to B) \in \mathcal{C}_\tau[\![\star_\sigma]\!]^j_\rho$. By the definition of the interpretation, it is enough to show that $\rho\,((x : k) \to B)$ is a closed type value of kind $\star_\sigma$. This is immediate by Lemma 5.5.7 and the definition of type values.

- Cases TArrComp, TNat, TSigma, TSum, TMu, TEq, and TEqT are similar to the previous case.

- $$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Gamma \vdash A : \star_\tau\end{array}}{\Gamma \vdash A : \star_\sigma}\ \textsc{TMonoPoly}$$

  In this case, according to the definition of the interpretation, we must show that $\rho\, A \leadsto^* V$ and $\cdot \vdash V : \star_\sigma$, which follows immediately by the IH for $\mathcal{E}_1$ and a use of TMonoPoly.

- Case TAt is similar to the previous case.

□

Normalization for closed terms is a direct corollary of the fundamental theorem of the interpretation.

Corollary 5.8.10 (Normalization)**.**

- If $\cdot \vdash^{\mathsf{L}} a : A$ then there is a value v such that $a \leadsto^* v$.

- If $\cdot \vdash A : k$ then there is a type value V such that $A \leadsto^* V$.

Proof. Since $\emptyset \models_j \cdot$ for any j by EnvNil, Theorem 5.8.9 yields $a \in \mathcal{C}^{\mathsf{L}}_{\mathsf{up}(A)}[\![A]\!]^j_\emptyset$ in the first case and $A \in \mathcal{C}_{\mathsf{up}(k)}[\![k]\!]^j_\emptyset$ in the second case. In both cases, unfolding the definition of computational interpretation directly yields the desired result. $\square$

The fundamental theorem also implies the consistency of the system, i.e., that there are uninhabited types.

Corollary 5.8.11 (Consistency)**.** There is no term a such that $\cdot \vdash^{\mathsf{L}} a : \mathsf{Z} = \mathsf{S}\,\mathsf{Z}$.

Proof. Suppose for a contradiction that $\cdot \vdash^{\mathsf{L}} a : \mathsf{Z} = \mathsf{S}\,\mathsf{Z}$. By EnvNil and Theorem 5.8.9, $a \in \mathcal{C}^{\mathsf{L}}_\tau[\![\mathsf{Z} = \mathsf{S}\,\mathsf{Z}]\!]^j_\emptyset$ for any j. Unfolding the definition of the interpretation, we find this implies that there is some term b such that $\mathsf{Z} \Rrightarrow^* b$ and $\mathsf{S}\,\mathsf{Z} \Rrightarrow^* b$. But examination of the definition of parallel reduction reveals that Z and $\mathsf{S}\,\mathsf{Z}$ step only to themselves, and b cannot be both—a contradiction. $\square$

In fact, as in LF$^\theta$, we can prove a more general result for both the term and type equalities. This will be convenient in the proof of progress:

Corollary 5.8.12 (Soundness of term equality)**.**
If $\cdot \vdash^{\mathsf{L}} a : b_1 = b_2$, then there exists some b such that $b_1 \Rrightarrow^* b$ and $b_2 \Rrightarrow^* b$.

Corollary 5.8.13 (Soundness of type equality)**.**
If $\cdot \vdash^{\mathsf{L}} a : A_1 = A_2$, then there exists some A such that $A_1 \Rrightarrow^* A$ and $A_2 \Rrightarrow^* A$.

Proof. By Theorem 5.8.9 and the definition of the interpretation. $\square$

5.9 Progress

Just as we saw in the case of LF$^\theta$, the proof of PCC$^\theta$'s progress lemma must wait until after we have demonstrated the consistency of the propositional equality. The reason is that the canonical forms lemmas would otherwise get stuck in the TConvT and TContra cases. Because of the unmarked @-type elimination rule EUnboxVal, the lemmas must be generalized a bit before we can prove them. Unlike in previous systems, where we generalized the canonical forms lemmas with respect to a syntactic equality, the presence of type level computation demands that we generalize with respect to convertibility or propositional equality (which we now know are the same in closed contexts).

Lemma 5.9.1 (Generalized canonical forms for Nat)**.** Suppose $\cdot \vdash^\theta v : A$ and $\cdot \vdash^{\mathsf{L}} b : A = \mathsf{Nat}$ or $\cdot \vdash^{\mathsf{L}} b : A = (...((\mathsf{Nat})@\theta_1)...)@\theta_n$ for some $\theta_1, ..., \theta_n$. Then either $v = \mathsf{Z}$ or $v = \mathsf{S}\,v'$ for some value v'.

Proof. By induction on the derivation $\mathcal{D} :: \cdot \vdash^{\theta} v : A$. Some cases do not apply because their subject is not a value. The EVAR case is ruled out because the context is empty. We consider the remaining cases:

- Suppose $\mathcal{D}$ goes by TCONTRA. Then there is a closed logical proof of some equality $A_1 = A_2$ such that A_1 and A_2 have distinct head forms. But by Corollary 5.8.13, A_1 and A_2 reduce to a common type—a contradiction, since reduction preserves head forms (Lemma 5.2.10), and that no type has two distinct head forms.

- If $\mathcal{D}$ goes by TZERO or TSUCC, the result is immediate as the subject has the desired form.

- If $\mathcal{D}$ goes by TUNBOXVAL or TMVAL, the result follows directly from the appropriate induction hypothesis.

- If $\mathcal{D}$ goes by TINL, then for some B_1 and B_2 we have either a derivation of

$$\cdot \vdash^{\mathsf{L}} b : (B_1 + B_2) = \mathsf{Nat}$$

  or of:

$$\cdot \vdash^{\mathsf{L}} b : (B_1 + B_2) = (...((\mathsf{Nat})@\theta_1)...)@\theta_n$$

  In either case, this is an equality between two types with different head forms. We may now obtain a contradiction by observing that Corollary 5.8.13 implies both sides of the relevant equality reduce to a common type, that reduction preserves head forms (Lemma 5.2.10), and no type has two distinct head forms.

- Cases EATINV, EARRINV1, EARRINV2, EARRINVT2, ESUMINV1, ESUMINV2, ESIGMAINV1, ESIGMAINV2, EMUINV, ELAMCOMP, ELAMPOLY, ERECCOMP, ERECPOLY, EPAIR, EINR, EROLL, EREFL, and EREFLT are similar to the previous case, since they all assign a type that has a head form that is not HNat or $\mathsf{HAt}\,\theta$ for some θ.

- $$\mathcal{D} = \dfrac{\cdot \vdash^{\mathsf{L}} b : b_1 = b_2 \qquad \overset{\mathcal{D}'}{\cdot \vdash^{\theta} v : [b_1/x]A} \qquad \cdot \vdash [b_2/x]A : \star_{\sigma}}{\cdot \vdash^{\theta} v : [b_2/x]A}\ \text{TCONV}$$

  The IH for $\mathcal{D}'$ yields the desired result, if we can show that $[b_1/x]A$ is provably equal to Nat or $(...((\mathsf{Nat})@\theta_1)...)@\theta_n$.

  Note that, by Lemma 5.8.12, there is some a such that $b_1 \Rrightarrow^* a$ and $b_2 \Rrightarrow^* a$. By Lemma 5.2.13, this means that $[b_1/x]A \Rrightarrow^* [a/x]A$ and $[b_2/x]A \Rrightarrow^* [a/x]A$.

  So, by two uses of EREFLT we have:

$$\cdot \vdash^{\mathsf{L}} \mathsf{trefl} : [b_1/x]A = [a/x]A \qquad \text{and} \qquad \cdot \vdash^{\mathsf{L}} \mathsf{trefl} : [a/x]A = [b_2/x]A$$

So, by ECONVT, we have:

$$\cdot \vdash^{\mathsf{L}} \mathsf{trefl} : [b_1/x]A = [b_2/x]A$$

And since we know that $[b_2/x]A$ is provably equal to a type of the appropriate form, another use of ECONVT yields the desired result.

- Case TCONVT is similar to the previous case.

□

Lemma 5.9.2 (Canonical forms for Nat)**.** If $\cdot \vdash^{\theta} v : \mathsf{Nat}$ then either $v = \mathsf{Z}$ or $v = \mathsf{S}\, v'$ for some value v'.

The other canonical forms lemmas are simple to prove with a similar generalization. We omit the details. The progress theorem is then an easy induction on the typing derivation:

Theorem 5.9.3 (Progress)**.** If $\cdot \vdash^{\theta} a : A$, then either a is a value or there exists a' such that $a \leadsto a'$.

Chapter 6

Difficulties Scaling Up

There's a monkey in the basement.
How did the monkey get there?
There's a monkey in the basement.
Where did the monkey come from?
Where did the monkey come from?
Where did the monkey come from?

The Monkey Song
The Mountain Goats

This thesis attempts to solve two somewhat unrelated problems with existing dependently typed languages. The combination of these two features in Theta is something of an accident.

The research presented here is an outgrowth of the Trellys project. Trellys is a collaborative research initiative to design a new, practical dependently typed language. Early Trellys languages had all the features of Theta and more. However, their metatheory proved intractable. Over time, the Trellys project splintered into smaller languages that solved more specific problems and added different restrictions to aid the metatheory. One such language is closely related to λ^{θ} [15], whose metatheory we explored in Chapter 3. This language stripped away every feature except the two fragments, and explored them in a simply typed setting.

When it came time to scale up, the simplest way to add dependent types was to add a primitive notion of equality [14], as we did with LF$^{\theta}$ in Chapter 4. We chose to use the generous, untyped and unmarked notion of equality that is described in this thesis (hereafter "Trellys equality"), inspired in large part by Guru [54], as a step towards the broader Trellys goal. Since the metatheory of LF$^{\theta}$ proved tractable and the Trellys equality did not introduce substantial metatheoretic complications, we were optimistic that the language and corresponding metatheory would scale gracefully to a complete core language with polymorphism and dependent types. Unfortunately, Trellys equality turns out to have problematic interactions with these features, when attempting to analyze it with standard metatheoretic techniques. At the same time as we were discovering this, we realized that the metatheory of large dependently typed core languages like Theta is not as well understood as we thought, even without the fragments and the unusual equality.

Thus, the system PCC$^\theta$ of Chapter 5 represents a kind of compromise. It develops new techniques to handle Trellys equality in the presence of type-level computation and polymorphism, but scales back on features like an infinite universe hierarchy and collapsed syntax, which are not completely explored even in the literature of Coq and Agda. We are proud of this achievement, but feel that it is also important to document the difficulties in bridging the gap between PCC$^\theta$ and Theta from a metatheoretic perspective. We will also explain some features we have considered but left out of Theta itself.

The goal of the present chapter, therefore, is to illustrate what makes the metatheory of full Theta interesting and challenging, highlighting the problems introduced by Trellys equality. It is important to emphasize that this is primarily a story of failed proof techniques and not of failed languages. We are not aware of any actual problems with Theta or the other languages we consider (except for the example in Section 6.2). Instead, this chapter will focus on what we have learned about the limits of known proof techniques, in the hope that other researchers tackling similar problems can learn from our efforts.

6.1 Programmatic Types

There is one feature we would like to include in Theta, but whose metatheory is so unusual we have left it out: programmatic types. In Theta and implicitly in PCC$^\theta$, types are required to check in the logical fragment. This restriction does not exist in LF$^\theta$, where instead the requirement is only that when a type is used in a given fragment, it must also check in that fragment.

Allowing programmatic terms to have programmatic types could be a useful feature. In the present systems, everything at type level and above must be defined using terminating recursion. Of course, the same arguments for the convenience of being allowed to program without this restriction apply regardless of whether we are writing terms or types, so it would nice to relax it. Sometimes potentially non-terminating types are explicitly useful—for example, we have used them in the context of generic programming in Agda by ignoring the termination checker [63].

However, allowing general recursion at the type level introduces substantial problems for the metatheory. In particular, the interpretation of types used in our normalization proofs must itself be a well-founded recursive function. Recursive functions cannot be interpreted directly in the way we have interpreted lambdas. It is tempting to think that the step-indexing technique we have used to handle *recursive types* could handle *recursive functions* at the type-level, but this is a misunderstanding of the technique, which captures a partial correctness property by counting reduction of terms, not of types. Indeed, it is not even clear what kind of partial correctness property we would want for the interpretation itself.

6.2 Extensionality

One advantage of our unmarked equality is that, in principle, we may add any (consistency) equality axioms we would like without creating stuck terms or ruining "canonicity". By contrast, when we add an equality axiom to Coq or Agda, it must be used explicitly via pattern matching or a `conv` eliminator, creating a term that cannot reduce. The equality inversion rules (like EAtInv) in PCC$^\theta$ are examples of convenient reasoning principles that do not interfere with reduction.

Since our equality has an extensional flavor, it is natural to consider adding an explicit axiom of functional extensionality. That is, to add a term of type:

```
(A B : Type) → (f g : A → B) → ((x:A) → f x = g x) → f = g
```

However, this axiom is actually *inconsistent* with full Theta.

The reason has to do with the unmarked function domains. Consider the functions `\y.0` and `\y.1`. These functions can be given many types in Theta because their domain is unspecified and the argument is not used in the body. One such type is `(Nat = Bool) → Nat`. According to the extensionality axiom listed above, to show these two functions are equal, it would be enough to show that:

```
(x:Nat = Bool) → ((\y.0) x) = ((\y.1) x)
```

But this type is actually inhabited by the term `\x.contra`, since anything may be proved from a contradictory equality via TContra.

Thus, with functional extensionality, it would be possible to obtain a proof of `\y.0 = \y.1`. Of course, these functions can also be given a type with an inhabited domain, like `Nat → Nat`. So, this equality would imply, for example, that `((\y.0) 2) = ((\y.1) 2)`. Thus, by TRefl and TConv, we could show `0 = 1`.

As far as we are aware, this contradiction only arises when a function is given a domain type that implies a contradiction. This suggests a potential solution: allow functional extensionality to be used only for functions whose domain is inhabited in the logical fragment. For example, we could rewrite the axiom above to:

```
(A B : Type) → (a : A@L) → (f g : A → B) → ((x:A) → f x = g x) → f = g
```

The extra premise `a : A@L` should rule out situations like the one above, since `Nat = Bool` is not inhabited in the logical fragment. Exploring the metatheory in the presence of such an axiom is an interesting direction for future work.

6.3 Infinite Universe Hierarchy

In full Theta, we have included an infinite hierarchy of universes `Type` ℓ. Such a hierarchy poses a substantial challenge for a normalization proof. The chief problem is that the introduction of such a hierarchy destroys our ability to identify the level of an expression just by examining it.

By way of context, recall the Calculus of Constructions, where there are only two universe levels $\star$ and $\Box$, such that $\star : \Box$. In this system, it is possible to prove a "classification" result of the following form:

If $\Gamma \vdash A : B$, then exactly one of the following holds:

- B is $\Box$. In this case, we call A a *kind*.
- $\Gamma \vdash B : \Box$. In this case, we call A a Γ-*constructor* (or a Γ-*type* in the special case where B is $\star$).
- $\Gamma \vdash B : \star$. In this case, we call A a Γ-*term*.

One convenient fact about this hierarchy is that it is possible to distinguish whether an expression A is a term, constructor, or kind simply by examining A itself (given a context to distinguish term variables from type variables). In particular, in the Calculus of Constructions we can easily show that the only "kinds" in the sense of the above definition are $\star$ and arrow-types that end in $\star$.

This fact is exploited in most proofs of normalization for CC. One key use is in defining the possible interpretations for types (e.g., the function $\mathsf{Cand}(k)$ from Chapter 5). Since types have kinds and kinds have a clear structure, we may define the possible interpretations of types by induction on the structure of kinds rather than considering arbitrary expressions.

But in a type system with an infinite hierarchy of universes, there is no "top level" like $\Box$ that gives us a foothold into characterizing the possible forms of types. Indeed, lambdas and applications live at every level, so defining $\mathsf{Cand}(k)$ is challenging.

Most existing normalization proofs skirt this problem in one of two ways. The proofs for systems that include datatypes and large eliminations include only one universe level, much like the Calculus of Constructions. These proofs include Werner's proof of normalization for a fragment of the Calculus of Inductive Constructions [64] and Goguen's proof of normalization for a fragment of UTT [28]. This is also the approach we have taken in Chapter 5.

On the other hand Luo's proof of normalization for the Extended Calculus of Constructions (ECC) includes a full predicative hierarchy [40]. Luo solves the problem of defining candidates by proving a *quasi-normalization* theorem before the main normalization theorem. This quasi-normalization result implies that expressions at the type level can be reduced to head forms, and thus the definition of candidates need not consider applications. Quasi-normalization is achieved by defining an intricate metric on types which Luo calls the "j-degree" of a type. It counts, roughly, the number of quantifications that occur at type level j in the type. Since a given expression can only use a finite part of the infinite hierarchy, it is then enough to show normalization for ECC^j, the system with only j levels. Since the top level of

ECC^j will have a structure much like the kind-level of CC, the standard techniques apply.

Unfortunately, a quasi-normalization approach will not work in our system. Luo's system includes a standard conversion rule for β-equality:

$$\frac{\Gamma \vdash a : A \qquad A \cong_\beta B}{\Gamma \vdash a : B}\ \textsc{TConv}$$

In order to handle this rule, Luo must show that j-degrees are preserved by conversion. That is, if $A \cong_\beta B$, then A and B have the same j-degree. Since Theta has a similar unmarked use of propositional equality, we would need to show that if $\Gamma \vdash^{\mathsf{L}} a : B_1 = B_2$, then B_1 and B_2 have the same j-degree. But this is not provable before we know the system is consistent—in general B_1 and B_2 may be completely unrelated.

6.4 Collapsed Syntax

Another major distinction between PCC$^\theta$ and Theta is that the syntax of Theta is collapsed, while the syntax of PCC$^\theta$ is stratified into terms, types and kinds. In some systems, this is a small distinction and it is simple to prove that well-typed terms in the collapsed syntax can be translated directly into a stratified syntax. However, this is not the case for Theta.

To illustrate why, let us consider what Theta would look like with if restricted to one universe level (like the Calculus of Constructions). For simplicity, we omit data types and induction.

$$\begin{array}{lcl} s & ::= & \star_\tau \mid \star_\sigma \mid \Box \\ \\ a,\ b,\ A,\ B & ::= & s \mid (x : A) \to B \mid a = b \mid A@\theta \\ & & \mid x \mid \lambda x.\, b \mid \mathsf{rec}\, f\ x.b \mid b\ a \mid \mathsf{refl} \mid \mathsf{abort} \mid \mathsf{contra} \mid \end{array}$$

Because we wish to consider only predicative polymorphism, we divide the standard sort $\star$ into $\star_\tau$ for monomorphic types and $\star_\sigma$ for polymorphic types, as we did in Chapter 5. Sort $\star_\tau$ has type $\Box$, but $\star_\sigma$ does not. This restriction, combined with the rule for arrow types, achieves predicative polymorphism. As in pure type systems, we use a set of "rules" $\mathcal{R}$ to classify the allowed arrow types.

$$\mathcal{R} = \{(\Box,\Box),\ (\star_\sigma,\Box),\ (\Box,\star_\sigma),\ (\star_\sigma,\star_\sigma),\ (\star_\tau,\star_\tau)\}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \star_\tau : \Box}\ \textsc{TStar} \qquad \frac{\Gamma \vdash^{\theta} A : \star_\tau}{\Gamma \vdash^{\theta} A : \star_\sigma}\ \textsc{TTauSig} \qquad \frac{\begin{array}{cc}\Gamma \vdash^{\mathsf{L}} A : s_1 & (s_1, s_2) \in \mathcal{R} \\ \Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} B : s_2 & \mathsf{Mob}\,(A)\end{array}}{\Gamma \vdash^{\mathsf{L}} (x : A) \to B : s_2}\ \textsc{TArr}$$

The rules $\mathcal{R}$ are standard except for the split sort $\star$. In particular, the first rule allows type-level computation (functions from types to types), the second rule allows dependent types (functions from terms to types), the third rule allows polymorphism (functions from types to terms), and the last two rules allow standard term-level functions. We see that, since $\star_\sigma$ does not have the type $\Box$, it cannot be used as the domain of a function. Since polymorphic function types themselves have kind $\star_\sigma$, they do not quantify over themselves, and the polymorphism is predicative.

It is not difficult to see that Trellys equality introduces challenges in translating the well-typed terms of this language into a stratified language like PCC$^\theta$. For example, consider the equality type $a = b$. In Theta, equalities like $3 = \mathsf{Bool}$ are well-typed, and even inhabited in the programmatic fragment. But in PCC$^\theta$ equalities must be between two terms or two types, so there is no way to translate this type. More, the presence of equalities like $\mathsf{Nat} = \star_\tau$ directly ruins the "classification" theorem from the previous section that we were attempting to regain by removing the universe hierarchy.

A natural approach to solving this problem is restricting the typing rules which create ambiguity in the levels of the system. For example, we might try adding restrictions to Theta's rule TEq to demand that equalities are always between expressions on the same "level". There are two problems with this approach. First, formulating such restrictions is surprisingly challenging (the reader is encouraged to try). Second, and more troubling, such an approach would break type safety. In particular, the arrow types $(x : A) \to B$ and $(x : A') \to B'$ may be on the same level even when A and A' are not. And as we saw in Chapter 4, type safety requires us to be able to show $A = A'$ whenever $(x : A) \to B = (x : A') \to B'$.

Despite these problems, because the use of the sorts $\star_\sigma$ and $\Box$ is carefully restricted, it is possible to set up the system so that the valid kinds may be distinguished. That is, in the system sketched here, if $\Gamma \vdash^\theta A : \Box$, then we can show A comes from the following grammar:

$$k ::= \star_\tau \mid (x : B) \to k \mid k@\theta$$

However, being able to characterize the kinds of the system turns out to be insufficient for the normalization proof—we also need the ability to distinguish between terms and types. Recall that the interpretation in Section 5.4 is indexed by an environment ρ which handles term variables and type variables differently. Term variables are mapped to a value, while type variables are mapped to a pair of a type and the interpretation of that type (roughly). In the presence of equalities like $\mathsf{Nat} = \star_\tau$, it may be the case that, for example, Bool has the type Nat. Thus, the interpretation runs the risk of treating a term like a type or vice versa. This creates a problem for lemmas involving environments, which need to know the difference (for example, Lemma 5.7.1).

6.5 Function Domains

Another difference between the language PCC$^\theta$ and core THETA is that function domains are annotated. This change makes it substantially easier to state the type interpretation used in the proof of normalization. In particular, recall the definition of the interpretation for type-level lambdas:

$$\mathcal{V}^{\theta}_{\pi}\,[\![\lambda x : k_1.B]\!]^{j}_{\rho} \qquad = (A \in \mathsf{TYP}, \mathcal{I} \in [\mathsf{Cand}(k_1)]^{(\theta\times\mathbb{N})}) \mapsto \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x\mapsto(A,\mathcal{I})]}$$

We interpret the lambda as a set-theoretic function. Here, we have given the set-theoretic function a domain that refers to the domain type of the lambda. It is surprisingly difficult to formulate an appropriate domain for this function without the kind k_1. One cannot simply use the set of all possible interpretations, since then the function being defined here will not be in that set.

One approach we considered in depth was to define the interpretation on typing derivations rather than expressions. Even if we removed the domain annotation from lambda expressions, the typing rule would still specify the domain:

$$\frac{\begin{array}{c}\Gamma, x : k_1 \vdash B : k_2 \\ \Gamma \vdash (x : k_1) \to k_2\end{array}}{\Gamma \vdash \lambda x : k_1.B : (x : k_1) \to k_2}\ \textsc{TLamTLC}$$

There are, however, several problems with working with derivations instead of expressions. First, it substantially complicates the metatheory, which is already somewhat complex. More pressingly, it is difficult to define an interpretation over derivations which has the necessary properties. To see why, recall the statement of the fundamental theorem of our interpretation:

Theorem. Suppose $\rho \models_j \Gamma$.

- If $\Gamma \vdash^{\theta_1} a : A$, then $\rho\ a \in \mathcal{C}^{\theta_1}_{\mathsf{up}(A)}[\![A]\!]^{j}_{\rho}$.
- If $\Gamma \vdash A : k$, then $\rho\ k \in \mathcal{C}_{\mathsf{up}(k)}[\![k]\!]^{j}_{\rho}$.

Consider the first clause of this theorem. If the interpretation is defined on typing derivations, we must come up with a derivation of $\Gamma \vdash A : k$ to use in the statement of the first clause. If we attempt to use a particular derivation, we will get stuck because the interpretations of polymorphic function types necessarily quantify over arbitrary derivations. So we must state the theorem to say that $\rho\ a$ is in the interpretation of *any* derivation of $\Gamma \vdash A : k$. Unfortunately, the details of such a proof require a coherence lemma of the form:

Lemma. If $\mathcal{D}, \mathcal{D}' :: \Gamma \vdash A : k$, then the interpretations of $\mathcal{D}$ and $\mathcal{D}'$ are the same.

This kind of lemma is standard when defining interpretations over typing derivations. However, because of the unmarked TConv and TContra rules in our system, there does not appear to be any appropriate interpretation of derivations that satisfies it.

It is worth pointing out that the problems described here only require us to annotate the domains of type-level lambdas whose domain is a kind. We have chosen to add domain annotations to other function forms in PCC$^\theta$ as well for uniformity, but they are not needed for the normalization proof.

6.6 Large Eliminations

One feature that appears quite straightforward to add to PCC$^\theta$ is *large eliminations.* For example, recall that system's rule for pattern matching on pairs:

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} a : (\Sigma x : A_1 \,.\, A_2)@\theta' \\ \Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a \vdash^{\theta} b : B \\ \Gamma \vdash B : \star_{\sigma}\end{array}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b : B}\ \textsc{EPCase}$$

This rule appears only at the term level. However, it would be convenient to be allowed to pattern match at the type level as well. We could add a type form $\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow B$ and a corresponding inference rule:

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : (\Sigma x : A_1 \,.\, A_2)@\theta' \\ \Gamma, x : A_1, y : A_2@\theta', z : \langle x, y\rangle = a \vdash B : k \\ \Gamma \vdash k\end{array}}{\Gamma \vdash \mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow B : k}\ \textsc{TPCase}$$

We require the scrutinee a to check in the logical fragment to preserve normalization of the type level, but otherwise the rule closely resembles its term-level counterpart. This new type form is a *large elimination* because it eliminates a term at the type level. But the rule TConv is already a kind of large elimination—it eliminates equality proofs at the type level. The machinery that is needed in the normalization proof to deal with this is already included in our proofs (primarily the environment ρ which maps terms to values). Thus, we could interpret this new type form as follows, in the logical fragment:

$$\mathcal{V}^{\theta}_{\pi}[\![\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow B]\!]^{j}_{\rho} \quad = \begin{cases} \mathcal{V}^{\theta}_{\pi}[\![B]\!]^{j}_{\rho[x \mapsto v_1][y \mapsto v_2][z \mapsto \mathsf{refl}]} & \text{If } \rho\, a \rightsquigarrow^{*} \langle v_1, v_2\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

It seems likely that, with the interpretation shown here, not much would need to change in the existing normalization proof for PCC$^\theta$.

Chapter 7

Related work

Shuffled up Sixth Street in the rain.
Kept my head down as I looked past the people.
And in the department store, I found what I was looking for.
This is the church, this is the crucible.

Wizard Buys a Hat
The Mountain Goats

This thesis attempts to solve two major problems with existing dependently typed languages. Unsurprisingly, both problems have been analyzed by other authors in the past. In this chapter, we examine some of the most closely related alternate approaches.

7.1 Other Approaches to Recursion and Partiality

Many authors have considered language features to model partiality and recursion in a consistent dependent type theory. Some of these techniques have even been implemented in existing dependently typed languages, with varying degrees of success.

7.1.1 Partiality Monad

Capretta proposed representing potentially non-terminating computations using a coinductive *partiality monad* [12]. This technique can be used in existing languages like Coq and Agda, which already support coinduction [20]. For example, Agda's partiality monad has been used to present subtyping for recursive types [24] and represent potentially infinite parsing trees [23].

In both Coq and Agda, coinduction is supported via the addition of several primitive operators. The Agda standard library includes the following specification:

```
postulate
  ∞  : ∀ {a} (A : Set a) → Set a
```

```
♯  : ∀ {a} {A : Set a} → A →∞ A
♭  : ∀ {a} {A : Set a} →∞ A → A
```

Here, **Set** is Agda's **Type** and a is a universe level, similar to ℓ in PCC$^\theta$. Intuitively, the ∞ type constructor creates a "suspended computation" of the appropriate type, while ♯ and ♭ are "delay" and "force" operators, respectively. For example, we may define infinite streams and a map function over them as follows:

```
data Stream (A : Set 0) : Set 0 where
  Cons : A →∞ (Stream A) → Stream A

map : {A B : Set 0} → (A → B) → Stream A → Stream B
map f (Cons x xs) = Cons (f x) (♯ (map f (♭ xs)))
```

There are several differences between this approach and the one outlined in this document. Coinduction is a very general method for representing infinite data, which we do not consider. On the other hand, corecursive definitions are required to be "productive". For example, the following stream definition is rejected by Agda:

```
loop : {A : Set 0} → Stream A
loop = loop
```

Productivity is approximated by the "guard condition", which requires every corecursive call to be "guarded" by a coinductive constructor. Thus, while corecursion is excellent for modeling specific cases of non-termination (like streams), it is not well-suited for modeling functions whose termination behavior we simply do not yet understand, or functions that we wish to prove terminating extrinsically.

Additionally, the PCC$^\theta$ approach has advantage that terminating and potentially partial functions are defined and reasoned about in the same way. As we can see above, working with coinduction in Agda requires the use of an alternate set of primitives, which demand their own operational semantics and require corresponding reasoning principles to be built into the language.

7.1.2 Non-Constructive Fixpoint semantics.

The work of Bertot and Komendantsky [8] describes a way to embed general recursive functions into Coq that does not use coinduction. They define a datatype partial A that is isomorphic to the usual Maybe A but is understood as representing a lifted CPO $A_\perp$, and use classical logic axioms to provide a fixpoint combinator fixp. When defining a recursive function the user must prove continuity side-conditions. Since recursive functions are defined nonconstructively they cannot be reduced directly, so instead one must reason about them using the fix-point equation.

7.1.3 Partial Types

Nuprl has at its core an untyped lambda calculus, capable of defining a general fixed point combinator for recursive computations. In the core type theory, all expressions must be proven terminating when used. Constable and Smith [19] integrated potentially nonterminating computations through the addition of a type $\overline{A}$ of partial terms of type A. The fixpoint operator then has type $(\overline{A} \to \overline{A}) \to \overline{A}$. However, to preserve the consistency of the logic, the type A must be restricted to *admissible* types. Crary [22] provides an expressive axiomatization of admissible types, but these conditions lead to significant proof obligations, especially when using Σ-types.

Smith [53] provides an example which shows that Nuprl needs this restriction. Writing $a\!\downarrow$ for "a terminates", define a Σ-type T of functions which are not total, and recursively define a p which inhabits T.

$$\begin{aligned}
\text{Total } (f : \mathbb{N} \to \overline{\mathbb{N}}) &\overset{\text{def}}{=} (n : \mathbb{N}) \to (f\ n)\!\downarrow \\
T &\overset{\text{def}}{=} \Sigma(f : \mathbb{N} \to \overline{\mathbb{N}}).\text{Total } f \to \text{False} \\
(p : T) &\overset{\text{def}}{=} \text{fix } (\lambda p.\langle g, \lambda h.\text{—}\rangle) \\
g &\overset{\text{def}}{=} \lambda x.\text{if } x = 0 \text{ then } 0 \text{ else } \pi_1(p)(x-1)
\end{aligned}$$

Here the dash is an (elided) proof which sneakily derives a contradiction using $\pi_2(p)$ and the hypothesis h that g is total. On the other hand, a separate induction shows that $\pi_1(p)$ *is* total; it returns 0 for all arguments. This is a contradiction.

PCC$^\theta$ has almost all the ingredients for this paradox. Instead of a recursively defined pair we can use a recursive function Unit $\to T$, and we can encode $a\downarrow$ as $\Sigma(y : A).a = y$. What saves us is that the proof in the second component of p uses the following reasoning principle: if $\pi_1(p)$ terminates, then p terminates. In Nuprl $a\downarrow$ is a primitive predicate and this inversion principle is built in. But using our encoding, a function $(\pi_1(p)\downarrow) \to (p\downarrow)$ would have to magically guess the second component of a pair knowing only the first component. If we assume this function as an axiom we can encode the paradox and derive inconsistency, so our consistency proof for PCC$^\theta$ shows that there is no way to write such a function.

7.1.4 Hoare Type Theory.

HTT [48, 56] is another embedding of general programs into a type theory like Coq. It goes beyond nontermination to also handle memory effects. Instead of a unary type constructor $\overline{A}$, it adds the indexed type $\{P\}x : A\{Q\}$ representing an effectful computation returning A and with pre- and postconditions P and Q. The assertions P and Q can use all of Coq, so the type of a computation can specify its behavior precisely. However, computations cannot be evaluated during typechecking (the fixpoint combinator and memory access primitives are implemented as Coq axioms with types but no reduction rules). Thus, as we observed with coinduction, potentially non-terminating expressions must be reasoned about with a whole new set of primitives.

7.1.5 Terminating Sublanguages

There are other dependently-typed languages which allow general recursion but identify a sublanguage of terminating expressions. Aura [34] and F* [57] do this using the kind system: expressions whose type has kind Prop are checked for normalization. Types can contain values but not non-value expressions, so there is no way to write separate proofs about programs. There also is no facility to treat programmatic values as proofs, e.g. a logical case expression cannot destruct a value from the nonterminating fragment.

ATS [16], Guru [54], and Sep^3 [35] are dependently-typed languages where the logical and programmatic fragments are syntactically separate—in effect rejecting the rule TSub. One of the gains of this separation is that the logical language can be made CBN even though the programmatic one is CBV, avoiding the need for thunking. To do inductive reasoning, the Sep^3 language adds an explicit "terminates" predicate.

Idris [11] is a full-spectrum dependently typed programming language that permits non-total definitions. Internally, it applies a syntactic test to check if function definitions are structurally decreasing, and programmers may ask whether particular definitions have been judged total. The typechecker will only reduce expressions that have been proved terminating, again precluding separate equational reasoning about partial programs. Idris' metatheory has not been studied formally.

7.1.6 The "Later" Modality

Nakano [47] introduced the "later" modality to define a total language with guarded recursive types. Intuitively, a term of type $\bullet A$ (pronounced "later A") will be usable as a term of type A in the future. The recursive type $\mu\, x.A$ then unfolds to $[\bullet\mu\, x.A/x]A$ rather than $[\mu\, x.A/x]A$. Using this modality, he is able to give the type $(\bullet A \to A) \to A$ to the Y combinator. This type allows programmers to define a variety of recursive functions while still ensuring that the language is normalizing. Nakano uses a step-indexed realizability interpretation to prove the normalization property for his language, suggesting deep connections with the present work.

The later modality has been used by subsequent authors to design languages for a variety of purposes. Krishnaswami and Benton use it define a total language for functional reactive programming [37, 36]. Birkedal et al. [9] study the topos of trees, which they observe can model an extension of Nakano's calculus to a full dependent type theory with guarded recursion. While these authors do not consider languages with partiality and their settings have substantial differences from our own, their success in extending step-indexing and closely related techniques to model recursion in larger languages demonstrates the versatility of the technique.

7.2 Modal Type Systems for Distributed Computation

Modal logics allow one to reason from multiple perspectives, called "possible worlds". It is tempting to view the language presented here as such a system, where the possible worlds are θ, the logical and computational fragments of the language.

One way to define a modal logic is to make the world explicit, for example using a judgement $\Gamma \vdash^{\theta} A$, stating that under the assumptions in Γ, the proposition A is true at the world θ. Each assumption in the context is tagged with the world where it holds:

$$\frac{(\theta, A) \in \Gamma}{\Gamma \vdash^{\theta} A}$$

In such as system, the at modality [33], internalizes the typing judgment into a proposition, with introduction form

$$\frac{\Gamma \vdash^{\theta'} A}{\Gamma \vdash^{\theta} A@\theta'}$$

and elimination form:

$$\frac{\Gamma \vdash^{\theta} A@\theta' \qquad \Gamma, (\theta', A) \vdash^{\theta} C}{\Gamma \vdash^{\theta} C}$$

Our mobile rule is similar to the perspective-shifting rule, called **get**, from ML5 [46, 45].

$$\frac{\Gamma \vdash^{\theta'} A \quad A \ \mathsf{mobile}}{\Gamma \vdash^{\theta} A}$$

This rule, shown above, allows a class of propositions to be directly translated between worlds. Here, mobile means almost the same thing as it does in PCC$^{\theta}$—in fact, ML5 inspired our choice of the name "mobile" for types whose values can freely move between the fragments. For example, base types (such as strings and integers) and the at modality ($A@\theta$) are always mobile, sums ($A + B$) are mobile only when their components (A and B) are mobile, but implications are never mobile.

One difference between our system and modal logics is our treatment of implication (i.e. function types). The functions in our system are required to have a "mobile" domain. Non-mobile types must be annotated with a fragment using the @ constructor. In modal logics, the domain and range of implications are in the same world. Such an approach is incompatible with our subsumption rule:

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A} \ \text{TSub}$$

Suppose A were a function type $B_1 \to B_2$ with no tag on the domain. When we defined such a function in the logical fragment, the function's body could make use of the fact that its argument checks logically. If the subsumption rule were used to transport the function to the programmatic fragment, it could be applied to terms that check only programmatically, potentially violating assumptions of its body.

7.3 Equality in Dependent Type Theory

Many previous authors have observed that the derived intensional equality in languages like Coq and Agda is restrictive and inconvenient to work with in practice. In this section we will examine other solutions to this problem.

7.3.1 John Major Equality

One problem with the traditional notion of equality is that it is homogeneous—both sides of an equality must have the same type. It is also possible to define a similar but heterogeneous notion of equality, known as "John Major" equality or **JMEq** [41]:

```
data JMEq : {A B : Type} →  A → B → Type where
  refl : {A : Type} → (a : A) → JMEq a a
```

Unfortunately, while **JMEq** allows the programmer to *state* any equality, these equalities can only be *used* when both sides have the same type. For example, in Coq, the derived elimination principle for **JMEq** has (roughly) the following type:

```
  (A : Type) → (x y : A) → (P : A → Set) → P x → JMeq x y → P y
```

This restriction on the elimination of equality proofs is often inconvenient. Massaging goals into a form where the equalities are usable often requires special tricks and idioms (see e.g. [17], Chapter 10).

7.3.2 Extensional Type Theory

Intensional type theories like Coq and Agda have two notions of equality. The *definitional* equality, which is typically β- or $\beta\eta-$ convertibility, is automatically used by the type system (much like PCC$^\theta$ equality), but cannot be explicitly manipulated by programmers:

$$\frac{A =_\beta B}{A \equiv B}\ \textsc{DefEq} \qquad \frac{\Gamma \vdash a : A \qquad A \equiv B}{\Gamma \vdash a : B}\ \textsc{Conv}$$

As we have seen, these languages also include a notion of *propositional* equality (which is often simply defined as a datatype). Propositional equality may be explicitly manipulated by the programmer, but its uses are marked in the syntax.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl} : \mathsf{Eq}\ A\ a\ a}\ \textsc{PropEq} \qquad \frac{\Gamma \vdash a : [b_1/x]A \qquad \Gamma \vdash p : \mathsf{Eq}\ B\ b_1\ b_2}{\Gamma \vdash \mathsf{conv}\ a\ \mathsf{by}\ p : [b_2/x]A}\ \textsc{PropConv}$$

On the other hand, in extensional type theories like Nuprl [18] the use of propositional equality is computationally irrelevant and unmarked in the syntax. This is achieved by *reflecting* propositional equality back into the definitional equality.

$$\frac{\Gamma \vdash a : \mathsf{Eq}\ A\ B_1\ B_2}{B_1 \equiv B_2}\ \textsc{Extensional}$$

This approach has some similarities with PCC$^\theta$. In particular, in extensional type theory, uses of conversion do not clutter terms. On the other hand, the extensional equality in languages like Nuprl is still defined in terms of a standard notion of propositional equality, with the associated limitations (for example, Nuprl's equality is homogeneous). The equational theory of extensional type theories is also somewhat different from PCC$^\theta$. For example, in Nuprl one can prove extensionality, which is incompatible with our language.

7.3.3 Observational Type Theory

Observational Type Theory [4] combines aspects of intensional and extensional type theory. Like intensional type theory, OTT distinguishes definitional and propositional equality. The propositional equality is then extended with a set of axioms which provide extensional reasoning principles. These axioms are equipped with operational semantics so that they do not block reduction. So, while conversion is still marked in terms, it is possible to prove equalities like ($a = \mathsf{conv}\ a\ \mathsf{by}\ b$).

Thus, the goals of OTT are very similar to our motivation for PCC$^\theta$'s equality. Both are heterogeneous notions of equality whose eliminations "clutter" terms less than the usual Coq or Agda equality. However, the two approaches are very different. While OTT attemps to achieve a more convenient equality by enhancing the type system with extensional axioms, the PCC$^\theta$ approach is to remove as much typing information as possible from terms.

7.3.4 Guru

Guru [54] includes a very operational notion of equality and was a major inspiration for PCC$^\theta$, as described in Chapter 6. Like PCC$^\theta$, Guru can eliminate equalities where the two sides have different types. Equalities are proved as in PCC$^\theta$ without any type-directed rules. However, unlike PCC$^\theta$, Guru's equality formation rule does not require that the equated expressions are even well-typed. This can be annoying in practice, because simple programmer errors are not caught by the type system.

7.3.5 GHC Core

GHC Core [55, 62] is similar to our language in not having separate notions of definitional and propositional equality. Instead, all type equivalences—which are implicit in Haskell source—must be justified by the typechecker by explicit proof terms. GHC lacks a consistent sublanguage corresponding to our logical fragment, so proofs are evaluated at runtime to ensure they are valid.

7.4 Step-indexed logical relations

Our proof technique draws heavily from previous work on step-indexed logical relations. The idea to approximate models of programming languages up to a number of remaining execution steps originated in the work of Appel and McAllester on foundational proof-carrying code [5]. They observed that the step indices allowed a natural interpretation of recursive types. Subsequently, Ahmed extended this technique to languages involving impredicative polymorphism, mutable state and other features [2, 1].

Hobor, Dockins and Appel have proposed a general *theory of indirection* which captures many of the common use-cases for step-indexed models [32]. They provide a general framework for applying these approximation techniques to resolve certain types of apparent circularity (similar to the problems with recursive types described above). In a recent draft [25], Dockins and Hobor have used this framework to provide a Hoare logic of total correctness for a small language with function pointers and semantic assertions. This work is closely related to the present development, but with different goals: they prove the soundness of a logic which can reason about termination, while we prove that every term in the logical fragment of our language terminates.

We first introduced the hybrid, partially step-indexed technique in the context of a simply typed language that resembles λ^{θ} [15]. In subsequent work, we extended this technique to handle dependent types in a setting similar to LF$^{\theta}$ [14].

Chapter 8

Conclusion

Our love is like Jesus, but worse.
Though you seal the cave up where you've lain its body,
It rises.
It rises.

Going to Marrakesh
The Extra Glenns

This thesis introduces a new dependently typed core language, PCC$^\theta$. The goal of PCC$^\theta$ is to solve two problems with existing dependently typed languages: the requirement of normalization makes programming with recursive functions difficult, and the standard notion of equality is insufficiently expressive and often inconvenient to work with. To solve the first problem, PCC$^\theta$ directly allows non-termination via general recursion, but identifies a terminating sublanguage, the "logical fragment". The type system of the logical fragment is consistent and may be used to reason about any PCC$^\theta$ program, even those that are potentially non-terminating. To solve the second problem, PCC$^\theta$ includes a new built-in notion of equality. This equality is heterogeneous and its uses are unmarked so they do not clutter expressions. Additionally, PCC$^\theta$ expressions include no typing annotations, so types do not get in the way of equalities.

At the outset of this project, we believed that most challenging parts of the metatheory of PCC$^\theta$ would relate to the inclusion of general recursion and the related interacting fragments. To study this, we created the language λ^θ, a simply typed variant. To prove normalization for the logical fragment of λ^θ we developed the technique of "partially step-indexed logical relations". Having demonstrated that the metatheory of a language with PCC$^\theta$-like logical and programmatic fragments is tractable, we added a PCC$^\theta$-like notion of equality to λ^θ to create the dependently typed language LF$^\theta$. The proof techniques we had previously developed scaled gracefully to this larger system. In both cases, the metatheory has been completely mechanized.

Bridging the gap between LF$^\theta$ and full PCC$^\theta$ proved more challenging. In particular, PCC$^\theta$ includes type-level computation and polymorphism. As we described in Chapter 6, these features interact in surprisingly subtle ways with PCC$^\theta$'s unmarked equality, creating substantial metatheoretic complications. Despite this, we were able to prove normalization for PCC$^\theta$, which combines our solutions to the problems of non-termination and equality reasoning with dependent types, type-level computation, and polymorphism. We used the techniques developed for λ^θ and LF$^\theta$, along with some new tricks, to show that terms in the logical fragment of PCC$^\theta$ normalize and that its type system is consistent.

It is natural to consider the plausibility of using the novel aspects of PCC$^\theta$ in a practical dependently typed language. Our solution to the problem of incorporating general recursion with dependent types seems like a relatively simple and safe addition to standard systems. Non-termination does introduce some problems for type inference, since comparing expressions becomes undecidable, but in practice this is not substantially different from the problem of comparing expressions when one might take a very long time to terminate. Since the fragments can be used independently, programmers who are satisfied with current dependently typed languages could work strictly in the logical fragment with minimal additional overhead. Additionally, programmers more familiar with languages like Haskell could program strictly in the programmatic fragment, and the ability to reason about their code would be available to them in the same language. Thus, such a system could be an excellent way to introduce more programmers to dependent types gradually.

The adoption of PCC$^\theta$'s equality seems trickier. First, extensional equality introduces substantial complications for type inference. Since uses of the propositional equality are not marked in terms, the type checker must decide where to insert them. In practice, some kind of annotation regime would likely be necessary. These problems are currently being explored by Vilhelm Sjöberg [51], and will be considered in depth in his upcoming thesis. A second concern is that the metatheory is murkier for a system with PCC$^\theta$'s equality that includes standard features like a collapsed syntax or a universe hierarchy. In practice, existing systems like Coq and Agda include features whose metatheory has not been studied in combination (or, in the case of Agda, at all), and we are not aware of any problems in THETA, the extended version of PCC$^\theta$. Still, we hope to continue studying the theory of THETA itself to gain additional confidence in the consistency of its logical fragment.

We believe that the solutions proposed in this thesis succeed in making dependently typed languages more usable. If PCC$^\theta$ is not precisely the dependently typed core language of the future, it certainly represents a useful step along the way.

Appendix A

Reduction Relations

For readability, we omitted the complete specification of parallel reduction in Chapter 4 and of several reduction relations in Chapter 5. Here we provide the missing details.

A.1 Parallel Reduction for LF$^\theta$

$$\boxed{a \Rrightarrow b}$$

$$\frac{}{a \Rrightarrow a}\ \textsc{PRefl}$$

$$\frac{a_1 \Rrightarrow a_1'}{\mathsf{ncase}_z\ \mathsf{Z}\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \Rrightarrow [\mathsf{refl}/z]a_1'}\ \textsc{PCaseZ}$$

$$\frac{v \Rrightarrow v' \quad a_2 \Rrightarrow a_2'}{\mathsf{ncase}_z\ \mathsf{S}\ v\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \Rrightarrow [\mathsf{refl}/z][v'/x]a_2'}\ \textsc{PCaseS}$$

$$\frac{v_1 \Rrightarrow v_1' \quad v_2 \Rrightarrow v_2' \quad b \Rrightarrow b'}{\mathsf{pcase}_z\ \langle v_1, v_2\rangle\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} \Rrightarrow [\mathsf{refl}/z][v_2'/y][v_1'/x]b'}\ \textsc{PCasePair}$$

$$\frac{v \Rrightarrow v' \quad a_1 \Rrightarrow a_1'}{\mathsf{scase}_z\ \mathsf{inl}\,v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \Rrightarrow [\mathsf{refl}/z][v'/x]a_1'}\ \textsc{PCaseInl}$$

$$\frac{v \Rrightarrow v' \quad a_2 \Rrightarrow a_2'}{\mathsf{scase}_z\ \mathsf{inr}\,v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \Rrightarrow [\mathsf{refl}/z][v'/x]a_2'}\ \textsc{PCaseInr}$$

$$\frac{v \Rrightarrow v' \quad a \Rrightarrow a'}{(\mathsf{rec}\ f\ x.a)\ v \Rrightarrow [v'/x][\mathsf{rec}\ f\ x.a'/f]a'}\ \textsc{PFun} \qquad \frac{v \Rrightarrow v' \quad a \Rrightarrow a'}{(\lambda x.a)\ v \Rrightarrow [v'/x]a'}\ \textsc{PLam}$$

$$\frac{v \Rrightarrow v' \quad a \Rrightarrow a'}{(\mathsf{ind}\ f\ x.a)\ v \Rrightarrow [v'/x][\lambda y.\lambda z.(\mathsf{ind}\ f\ x.a')\ y/f]a'}\ \textsc{PInd}$$

$$\frac{v \Rrightarrow v'}{\mathsf{unroll}\,(\mathsf{roll}\ v) \Rrightarrow v'}\ \textsc{PUnroll}$$

$$\frac{a \Rrightarrow a'}{\mathsf{rec}\ f\ x.a \Rrightarrow \mathsf{rec}\ f\ x.a'}\ \textsc{PFun1} \qquad \frac{a \Rrightarrow a'}{\lambda x.a \Rrightarrow \lambda x.a'}\ \textsc{PLam1}$$

$$\frac{a \Rrightarrow a'}{\mathsf{ind}\ f\ x.a \Rrightarrow \mathsf{ind}\ f\ x.a'}\ \textsc{PInd1} \qquad \frac{a \Rrightarrow b}{\mathsf{S}\ a \Rrightarrow \mathsf{S}\ b}\ \textsc{PSucc1}$$

$$\frac{a \Rrightarrow a' \quad a_1 \Rrightarrow a_1' \quad a_2 \Rrightarrow a_2'}{\mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \Rrightarrow \mathsf{ncase}_z\ a'\ \mathsf{of}\ \{Z \Rightarrow a_1'; \mathsf{S}\ x \Rightarrow a_2'\}}\ \textsc{PNCase1}$$

$$\frac{a \Rrightarrow b}{\mathsf{inl}\ a \Rrightarrow \mathsf{inl}\ b}\ \textsc{PInl1} \qquad \frac{a \Rrightarrow b}{\mathsf{inr}\ a \Rrightarrow \mathsf{inr}\ b}\ \textsc{PInr1} \qquad \frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{\langle a, b\rangle \Rrightarrow \langle a', b'\rangle}\ \textsc{PPair1}$$

$$\frac{a \Rrightarrow a' \quad a_1 \Rrightarrow a_1' \quad a_2 \Rrightarrow a_2'}{\mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \Rrightarrow \mathsf{scase}_z\ a'\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1'; \mathsf{inr}\ x \Rightarrow a_2'\}}\ \textsc{PSC1}$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{\mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} \Rrightarrow \mathsf{pcase}_z\ a'\ \mathsf{of}\ \{(x,\ y) \Rightarrow b'\}}\ \textsc{PPCase1}$$

$$\frac{a \Rrightarrow b}{\mathsf{roll}\ a \Rrightarrow \mathsf{roll}\ b}\ \textsc{PRoll1} \qquad \frac{a \Rrightarrow b}{\mathsf{unroll}\ a \Rrightarrow \mathsf{unroll}\ b}\ \textsc{PUnroll1}$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{a\ b \Rrightarrow a'\ b'}\ \textsc{PApp1} \qquad \frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{(x\!:\!A) \to B \Rrightarrow (x\!:\!A') \to B'}\ \textsc{PArr1}$$

$$\frac{A \Rrightarrow A'}{A@\theta \Rrightarrow A'@\theta}\ \text{PAt1} \qquad \frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{A + B \Rrightarrow A' + B'}\ \text{PSum1}$$

$$\frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{\Sigma x{:}A.B \Rrightarrow \Sigma x{:}A'.B'}\ \text{PSigma1}$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{a = b \Rrightarrow a' = b'}\ \text{PEq1} \qquad \frac{A \Rrightarrow A'}{\mu x.A \Rrightarrow \mu x.A'}\ \text{PMu1}$$

$$\boxed{a \Rrightarrow^* b}$$

$$\frac{}{a \Rrightarrow^* a}\ \text{MPRefl} \qquad \frac{\begin{array}{c} a \Rrightarrow b \\ b \Rrightarrow^* b' \end{array}}{a \Rrightarrow^* b'}\ \text{MPStep}$$

A.2 Reduction for PCC$^\theta$

A.2.1 Deterministic reduction

$$\boxed{a \leadsto b}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{Z}\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \leadsto [\mathsf{refl}/z]a_1}\ \text{SCaseZ}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{S}\,v\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \leadsto [\mathsf{refl}/z][v/x]a_2}\ \text{SCaseS}$$

$$\frac{}{\mathsf{pcase}_z\ \langle v_1, v_2\rangle\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b \leadsto [v_1/x][v_2/y][\mathsf{refl}/z]b}\ \text{SCaseP}$$

$$\frac{}{\mathsf{scase}_z\ \mathsf{inl}\,v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \leadsto [\mathsf{refl}/z][v/x]a_1}\ \text{SCaseL}$$

$$\frac{}{\mathsf{scase}_z\ \mathsf{inr}\,v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \leadsto [\mathsf{refl}/z][v/x]a_2}\ \text{SCaseR}$$

$$\frac{}{(\lambda x\!:\!A.b)\ v \leadsto [v/x]b}\ \textsc{SBeta} \qquad \frac{}{(\lambda x\!:\!k.b)\ V \leadsto [V/x]b}\ \textsc{SBetaT}$$

$$\frac{}{(\mathsf{rec}\, f\ (x\!:\!A).b)\ v \leadsto [v/x][\mathsf{rec}\, f\ (x\!:\!A).b/f]b}\ \textsc{SFBeta}$$

$$\frac{}{(\mathsf{rec}\, f\ (x\!:\!k).b)\ V \leadsto [V/x][\mathsf{rec}\, f\ (x\!:\!k).b/f]b}\ \textsc{SFBetaT}$$

$$\frac{}{\mathsf{unroll}\,(\mathsf{roll}\, v) \leadsto v}\ \textsc{SUnroll}$$

$$\frac{a \leadsto a'}{\mathsf{S}\ a \leadsto \mathsf{S}\ a'}\ \textsc{SSucc1} \qquad \frac{a \leadsto a'}{a\ b \leadsto a'\ b}\ \textsc{SApp1} \qquad \frac{b \leadsto b'}{v\ b \leadsto v\ b'}\ \textsc{SApp2}$$

$$\frac{a \leadsto a'}{a\ B \leadsto a'\ B}\ \textsc{SAppT1} \qquad \frac{B \leadsto B'}{v\ B \leadsto v\ B'}\ \textsc{SAppT2}$$

$$\frac{a \leadsto a'}{\mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \leadsto \mathsf{ncase}_z\ a'\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\}}\ \textsc{SNCase1}$$

$$\frac{a \leadsto b}{\mathsf{inl}\, a \leadsto \mathsf{inl}\, b}\ \textsc{SInl1} \qquad \frac{a \leadsto b}{\mathsf{inl}\, a \leadsto \mathsf{inl}\, b}\ \textsc{SInl1}$$

$$\frac{a \leadsto a'}{\mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \leadsto \mathsf{scase}_z\ a'\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\}}\ \textsc{SSC1}$$

$$\frac{a \leadsto a'}{\langle a, b\rangle \leadsto \langle a', b\rangle}\ \textsc{SPair1} \qquad \frac{b \leadsto b'}{\langle v, b\rangle \leadsto \langle v, b'\rangle}\ \textsc{SPair2}$$

$$\frac{a \leadsto a'}{\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b \leadsto \mathsf{pcase}_z\ a'\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b}\ \textsc{SPCase1}$$

$$\frac{a \leadsto a'}{\mathsf{roll}\, a \leadsto \mathsf{roll}\, a'}\ \textsc{SRoll1} \qquad \frac{a \leadsto a'}{\mathsf{unroll}\, a \leadsto \mathsf{unroll}\, a'}\ \textsc{SUnroll1}$$

$$\boxed{A \leadsto B}$$

$$\frac{}{(\lambda x : A.B)\ v \leadsto [v/x]B}\ \textsc{TSBeta} \qquad \frac{}{(\lambda x : k.B)\ V \leadsto [V/x]B}\ \textsc{TSBetaT}$$

$$\frac{A \leadsto A'}{A\ b \leadsto A'\ b}\ \textsc{TSApp1} \qquad \frac{b \leadsto b'}{V\ b \leadsto V\ b'}\ \textsc{TSApp2}$$

$$\frac{A \leadsto A'}{A\ B \leadsto A'\ B}\ \textsc{TSAppT1} \qquad \frac{B \leadsto B'}{V\ B \leadsto V\ B'}\ \textsc{TSAppT2}$$

$$\boxed{a \leadsto^{j} b}$$

$$\frac{}{a \leadsto^{0} a}\ \textsc{MSRefl} \qquad \frac{\begin{array}{c} a \leadsto b \\ b \leadsto^{j} b' \end{array}}{a \leadsto^{1+j} b'}\ \textsc{MSStep}$$

A.2.2 Parallel reduction

$$\boxed{a \Rrightarrow b}$$

$$\frac{}{a \Rrightarrow a}\ \textsc{PRefl}$$

$$\frac{a_1 \Rrightarrow a_1'}{\mathsf{ncase}_z\ \mathsf{Z}\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \Rrightarrow [\mathsf{refl}/z]a_1'}\ \textsc{PCaseZ}$$

$$\frac{v \Rrightarrow v' \quad a_2 \Rrightarrow a_2'}{\mathsf{ncase}_z\ \mathsf{S}\,v\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \Rrightarrow [v'/x][\mathsf{refl}/z]a_2'}\ \textsc{PCaseS}$$

$$\frac{v \Rrightarrow v' \quad a_1 \Rrightarrow a_1'}{\mathsf{scase}_z\ \mathsf{inl}\,v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \Rrightarrow [v'/x][\mathsf{refl}/z]a_1'}\ \textsc{PCaseInl}$$

$$\frac{v \Rrightarrow v' \quad a_2 \Rrightarrow a_2'}{\mathsf{scase}_z\ \mathsf{inr}\,v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \Rrightarrow [v'/x][\mathsf{refl}/z]a_2'}\ \textsc{PCaseInr}$$

$$\frac{v_1 \Rrightarrow v_1' \quad v_2 \Rrightarrow v_2' \quad b \Rrightarrow b'}{\mathsf{pcase}_z\ \langle v_1, v_2\rangle\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b \Rrightarrow [v_1'/x][v_2'/y][\mathsf{refl}/z]b'}\ \textrm{PC}\textsc{ase}\textrm{P}$$

$$\frac{v \Rrightarrow v' \quad b \Rrightarrow b'}{(\lambda x\!:\!A.b)\ v \Rrightarrow [v'/x]b'}\ \textrm{PB}\textsc{eta} \qquad \frac{V \Rrightarrow V' \quad b \Rrightarrow b'}{(\lambda x\!:\!k.b)\ V \Rrightarrow [V'/x]b'}\ \textrm{PB}\textsc{eta}\textrm{T}$$

$$\frac{v \Rrightarrow v' \quad a \Rrightarrow a'}{(\mathsf{rec}\, f\ (x\!:\!A).a)\ v \Rrightarrow [v'/x][\mathsf{rec}\, f\ (x\!:\!A).a'/f]a'}\ \textrm{PFB}\textsc{eta}$$

$$\frac{V \Rrightarrow V' \quad a \Rrightarrow a'}{(\mathsf{rec}\, f\ (x\!:\!k).a)\ V \Rrightarrow [V'/x][\mathsf{rec}\, f\ (x\!:\!k).a'/f]a'}\ \textrm{PFB}\textsc{eta}\textrm{T}$$

$$\frac{v \Rrightarrow v'}{\mathsf{unroll}\,(\mathsf{roll}\, v) \Rrightarrow v'}\ \textrm{PU}\textsc{nroll}$$

$$\frac{b \Rrightarrow b'}{\lambda x\!:\!A.b \Rrightarrow \lambda x\!:\!A.b'}\ \textrm{PL}\textsc{am}1 \qquad \frac{b \Rrightarrow b'}{\lambda x\!:\!k.b \Rrightarrow \lambda x\!:\!k.b'}\ \textrm{PL}\textsc{am}\textrm{T}1$$

$$\frac{a \Rrightarrow a'}{\mathsf{rec}\, f\ (x\!:\!A).a \Rrightarrow \mathsf{rec}\, f\ (x\!:\!A).a'}\ \textrm{PF}\textsc{un}1 \qquad \frac{a \Rrightarrow a'}{\mathsf{rec}\, f\ (x\!:\!k).a \Rrightarrow \mathsf{rec}\, f\ (x\!:\!k).a'}\ \textrm{PF}\textsc{un}\textrm{T}1$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{a\ b \Rrightarrow a'\ b'}\ \textrm{PA}\textsc{pp}1 \qquad \frac{a \Rrightarrow a' \quad B \Rrightarrow B'}{a\ B \Rrightarrow a'\ B'}\ \textrm{PA}\textsc{pp}\textrm{T}1$$

$$\frac{a \Rrightarrow a'}{\mathsf{S}\ a \Rrightarrow \mathsf{S}\ a'}\ \textrm{PS}\textsc{ucc}1$$

$$\frac{a \Rrightarrow a' \quad b_1 \Rrightarrow b_1' \quad b_2 \Rrightarrow b_2'}{\mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow b_1; \mathsf{S}\ x \Rightarrow b_2\} \Rrightarrow \mathsf{ncase}_z\ a'\ \mathsf{of}\ \{Z \Rightarrow b_1'; \mathsf{S}\ x \Rightarrow b_2'\}}\ \textrm{PNC}\textsc{ase}1$$

$$\frac{a \Rrightarrow a' \quad b_1 \Rrightarrow b_1' \quad b_2 \Rrightarrow b_2'}{\mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1; \mathsf{inr}\ x \Rightarrow b_2\} \Rrightarrow \mathsf{scase}_z\ a'\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1'; \mathsf{inr}\ x \Rightarrow b_2'\}}\ \textrm{PSC}1$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{\mathsf{pcase}_z\ a\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b \Rrightarrow \mathsf{pcase}_z\ a'\ \mathsf{of}\ \langle x, y\rangle\ \Rightarrow b'}\ \text{PPCase1}$$

$$\frac{a \Rrightarrow a'}{\mathsf{inl}\ a \Rrightarrow \mathsf{inl}\ a'}\ \text{PInl1} \qquad \frac{a \Rrightarrow a}{\mathsf{inr}\ a \Rrightarrow \mathsf{inr}\ a'}\ \text{PInr1} \qquad \frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{\langle a, b\rangle \Rrightarrow \langle a', b'\rangle}\ \text{PPair1}$$

$$\frac{a \Rrightarrow a'}{\mathsf{roll}\ a \Rrightarrow \mathsf{roll}\ a'}\ \text{PRoll1} \qquad \frac{a \Rrightarrow a'}{\mathsf{unroll}\ a \Rrightarrow \mathsf{unroll}\ a'}\ \text{PUnroll1}$$

$$\boxed{A \Rrightarrow B}$$

$$\frac{}{A \Rrightarrow A}\ \text{PTRefl}$$

$$\frac{v \Rrightarrow v' \quad B \Rrightarrow B'}{(\lambda x : A.B)\ v \Rrightarrow [v'/x]B'}\ \text{PTBeta} \qquad \frac{V \Rrightarrow V' \quad B \Rrightarrow B'}{(\lambda x : k.B)\ V \Rrightarrow [V'/x]B'}\ \text{PTBetaT}$$

$$\frac{B \Rrightarrow B'}{\lambda x : A.B \Rrightarrow \lambda x : A.B'}\ \text{PTLam1} \qquad \frac{B \Rrightarrow B'}{\lambda x : k.B \Rrightarrow \lambda x : k.B'}\ \text{PTLamT1}$$

$$\frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{(x : A) \to B \Rrightarrow (x : A') \to B'}\ \text{PTArr1}$$

$$\frac{k \Rrightarrow k' \quad B \Rrightarrow B'}{(x : k) \to B \Rrightarrow (x : k') \to B'}\ \text{PTArrT1}$$

$$\frac{A \Rrightarrow A' \quad b \Rrightarrow b'}{A\ b \Rrightarrow A'\ b'}\ \text{PTApp1} \qquad \frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{A\ B \Rrightarrow A'\ B'}\ \text{PTAppT1}$$

$$\frac{A \Rrightarrow A'}{A@\theta \Rrightarrow A'@\theta}\ \text{PTAt1} \qquad \frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{A + B \Rrightarrow A' + B'}\ \text{PTSum1}$$

$$\frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{\Sigma x{:}A\,.\,B \Rrightarrow \Sigma x{:}A'\,.\,B'}\ \text{PTSigma1} \qquad \frac{A \Rrightarrow A'}{\mu\, x.A \Rrightarrow \mu\, x.A'}\ \text{PTMu1}$$

$$\frac{a \Rrightarrow a' \quad b \Rrightarrow b'}{a = b \Rrightarrow a' = b'}\ \textsc{PTEq1} \qquad \frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{A = B \Rrightarrow A' = B'}\ \textsc{PTEqT1}$$

$$\boxed{k_1 \Rrightarrow k_2}$$

$$\frac{}{k_1 \Rrightarrow k_2}\ \textsc{PKRefl}$$

$$\frac{A \Rrightarrow A' \quad k_2 \Rrightarrow k_2'}{(x : A) \to k_2 \Rrightarrow (x : A') \to k_2'}\ \textsc{PKArr1}$$

$$\frac{k_1 \Rrightarrow k_1' \quad k_2 \Rrightarrow k_2'}{(x : k_1) \to k_2 \Rrightarrow (x : k_1') \to k_2'}\ \textsc{PKArrT1}$$

$$\boxed{a \Rrightarrow^* b}$$

$$\frac{}{a \Rrightarrow^* a}\ \textsc{MPRefl} \qquad \frac{\begin{array}{l} a \Rrightarrow b \\ b \Rrightarrow^* b' \end{array}}{a \Rrightarrow^* b'}\ \textsc{MPStep}$$

$$\boxed{A \Rrightarrow^* B}$$

$$\frac{}{A \Rrightarrow^* A}\ \textsc{MPTRefl} \qquad \frac{\begin{array}{l} A \Rrightarrow B \\ B \Rrightarrow^* B' \end{array}}{A \Rrightarrow^* B'}\ \textsc{MPTStep}$$

Bibliography

[1] Amal Ahmed. *Semantics of Types for Mutable State.* PhD thesis, Princeton University, 2004.

[2] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP '06: Proceedings of the 15th European Symposium on Programming*, 2006.

[3] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[4] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming Languages meets Program Verification*, pages 57–68. ACM, 2007.

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[6] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[7] B. Barras and B. Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

[8] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in coq. In *Proceedings of the 10th international ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '08, pages 89–96, New York, NY, USA, 2008. ACM.

[9] Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *LICS 2011*, pages 55–64, June 2011.

[10] Edwin Brady and Kevin Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102(2):145–176, 2010.

[11] Edwin C. Brady. Idris—systems programming meets full dependent types. In *PLPV'11: Programming languages meets program verification*, pages 43–54. ACM, 2011. ISBN 978-1-4503-0487-0.

[12] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[13] Chris Casinghino. Combining proofs and programs: Digital appendix. 2014. Available from `http://www.seas.upenn.edu/~ccasin/papers/thesis-appendix.tgz`.

[14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL, 2014*, pages 33–45.

[15] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Step-indexed normalization for a language with general recursion. In James Chapman and Paul Blain Levy, editors, *MSFP '12: Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 25–39. Open Publishing Association, 2012.

[16] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 66–77, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

[17] Adam Chlipala. *Certified Programming with Dependent Types*, 2011. URL `http://adam.chlipala.net/cpdt/`.

[18] Robert Constable and the PRL group. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[19] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, LICS '87, pages 183–193, 1987.

[20] Thierry Coquand. Infinite objects in type theory. In *Proceedings of the international workshop on Types for proofs and programs*, pages 62–78, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc. ISBN 3-540-58085-9.

[21] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Compututation*, 76(2-3):95–120, 1988.

[22] Karl Crary. *Type Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[23] Nils Anders Danielsson. Total parser combinators. In *ICFP, 2010*, pages 285–296, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

[24] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Jules Desharnais, and BÃľchir Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-13320-6.

[25] Robert Dockins and Aquinas Hobor. A theory of termination via indirection. In Amal Ahmed, Nick Benton, Lars Birkedal, and Martin Hofmann, editors, *Modelling, Controlling and Reasoning About State*, number 10351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[26] Herman Geuvers. A short and flexible proof of Strong Normalization for the Calculus of Constructions. In *TYPES '94*, volume 996 of *LNCS*, pages 14–38, 1995.

[27] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[28] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

[29] Georges Gonthier. A computer-checked proof of the four-colour theorem. 2005. Available at `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[30] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[31] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:194–204, 1993.

[32] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *POPL, 2010*, pages 171–185, 2010.

[33] Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). In *ESOP'04: European Symposium on Programming*, volume 2986 of *LNCS*, pages 219–233. Springer, 2004.

[34] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: A programming language for authorization and audit. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, 2008.

[35] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV '12: Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, 2012.

[36] Neelakantan Krishnaswami and Nick Benton. A semantic model for graphical user interfaces. In *ICFP, 2011*, pages 45–57, New York, NY, USA, 2011. ACM.

[37] Neelakantan Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *LICS, 2011*, pages 257–266, June 2011.

[38] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[39] Daniel R. Licata and Robert Harper. Positively dependent types. In *PLPV '09: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, pages 3–14, New York, NY, USA, 2008. ACM.

[40] Zhaohui Luo. *Computation and reasoning: a type theory for computer science.* Oxford University Press, New York, NY, USA, 1994.

[41] Conor McBride. Elimination with a Motive. In *Types for Proofs and Programs: International Workshop (TYPES 2000)*, volume 2277 of *LNCS*, pages 197–216. Springer, 2002.

[42] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[43] Alexandre Miquel. The implicit calculus of constructions - extending pure type systems with an intersection type binder and subtyping. In *TLCA '01: Proceeding of 5th international conference on Typed Lambda Calculi and Applications*, volume 2044 of *LNCS*, pages 344–359. Springer, 2001.

[44] Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently-typed programming. In *International Conference on Functional Programming*, 2010.

[45] Tom Murphy, VII. *Modal Types for Mobile Code.* PhD thesis, Carnegie Mellon, January 2008. URL `http://tom7.org/papers/`. Available as technical report CMU-CS-08-126.

[46] Tom Murphy, VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, 2007.

[47] Hiroshi Nakano. A modality for recursion. In *LICS, 2000*, pages 255–, Washington, DC, USA, 2000. IEEE Computer Society.

[48] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP '08: International Conference on Functional Programming*, pages 229–240. ACM, 2008.

[49] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[50] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[51] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In submission, 2014.

[52] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In James Chapman and Paul Blain Levy, editors, *MSFP '12: Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 112–162. Open Publishing Association, 2012.

[53] Scott Fraser Smith. *Partial Objects in Type Theory.* PhD thesis, Cornell University, 1988.

[54] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in guru. In *PLPV '09: Proceedings of the 3rd workshop on Programming Languages meets Program Verification*, pages 49–58, 2009.

[55] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton-Jones, and Kevin Donnelly. System f with type equality coercions. In *TLDI 07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in Languages Design and Implementation*, pages 53–66. ACM, 2007.

[56] Kasper Svendsen, Lars Birkedal, and Aleksandar Nanevski. Partiality, state and dependent types. In *Typed Lambda Calculi and Applications (TLCA'11)*, volume 6690 of *LNCS*, pages 198–212. Springer, 2011.

[57] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 285–296. ACM, 2011.

[58] William W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967.

[59] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.3*. LogiCal Project, 2010. Available at `http://coq.inria.fr/V8.3/refman/`.

[60] Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polytypic properties and proofs in Coq. In *WGP '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, pages 1–12, New York, NY, USA, 2009. ACM.

[61] Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Formal polytypic programs and proofs. *Journal of Functional Programming*, 20:213–270, 2010.

[62] Dimitrios Vytiniotis and Simon Peyton-Jones. Practical aspects of evidence-based compilation in System FC, 2011. Unpublished.

[63] Stephanie Weirich and Chris Casinghino. Arity-Generic Datatype-Generic Programming. In *PLPV '10: Proceedings of the 4th Workshop on Programming Languages Meets Program Verification*, 2010.

[64] Benjamin Werner. *Une théorie des constructions inuductives*. PhD thesis, Université Paris VII, 1994.

[65] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.